Algebraic Techniques for XML View Maintenance

ANGELA BONIFATI, Lille 1 University and Inria Links (France)
MARTIN GOODFELLOW, University of Strathclyde (UK)
IOANA MANOLESCU, Inria Oak (France)
DOMENICA SILEO, University of Basilicata (Italy)

Materialized views can bring important performance benefits when querying XML documents. In the presence of XML document changes, materialized views need to be updated to faithfully reflect the changed document. In this work, we present an algebraic approach for propagating source updates to XML materialized views expressed in a powerful XML tree pattern formalism. Our approach differs from the state of the art in the area in two important ways. First, it relies on set-oriented, algebraic operations, to be contrasted with node-based previous approaches. Second, it exploits state-of-the-art features of XML stores and XML query evaluation engines, notably XML structural identifiers and associated structural join algorithms. We present algorithms for determining how updates should be propagated to views, and highlight the benefits of our approach over existing algorithms through a series of experiments.

General Terms: Design, Algorithms, Experimentation, Performance

Additional Key Words and Phrases: XML view maintenance, XML updates, XML query processing.

1. INTRODUCTION

XML data management has reached by now a certain level of maturity, with many commercial and open-source systems supporting the W3C's XPath and XQuery [XQuery 1.0 2009] standards for querying XML documents. The complexity of XPath, XQuery and of XML data itself has raised many performance challenges. One direction of work towards improving the performance of XML query evaluation consists of relying on materialized views (or caches) storing pre-computed query results, based on which queries can be answered faster than by using the original documents only [Balmin et al. 2004a; El-Sayed et al. 2006; Mandhani and Suciu 2005; Onose et al. 2006; Xu and Ozsoyoglu 2005]. Such techniques have been shown to improve query evaluation performance by up to several orders of magnitude.

More recently, the W3C has also proposed an update extension to the XQuery language, namely XQuery Update [XQuery Update Facility 1.0 2009]. XQuery Update is gradually being implemented in XML data management platforms. When materialized views are used as a performance-enhancing tool, updates to the XML database raise two new problems. First, one has to determine whether the result of a view should change due to the update (or, as often said, whether the update *affects* the view). This problem has been studied recently in [Benedikt and Cheney 2010; Bidoit et al. 2010], and in the particular case when XML schemas are available to describe the documents in [Benedikt et al. 2005]. Second, when a view is indeed affected, a related issue is how to efficiently update the view to reflect the update. This second problem is the main focus of this paper.

Figure 1 illustrates the view maintenance problem in this context. Evaluating the view v over the XML document d leads to materializing $v(d)$. An XML update transforms d into d', and correspondingly the affected view v should be transformed into $v(d')$. One possibility is to evaluate v from scratch on the modified document d'. Instead, our focus is on incrementally modifying v by adding, removing, or modifying data as needed, to transform it into $v(d')$, without recomputing it.

The incremental maintenance of XML materialized views has been considered in previous works [Björklund et al. 2009; Choi et al. 2008; Dimitrova et al. 2003; El-Sayed et al. 2006; Foster et al. 2008; Onizuka et al. 2005; Sawires et al. 2005]. Maintaining XML views over relational databases is studied in [Choi et al. 2008]. The maintenance of boolean XPath queries is studied in [Björklund et al. 2009]. Views expressed in a richer XPath dialect are considered in [Sawires et al. 2005; Sawires et al. 2006], which focus on *node-level updates*, that is, they consider updates which add or remove exactly one node to/from the document. Node-level updates are propagated to XQuery views in [Dimitrova et al. 2003]. While node-level updates are conceptually simple, updates in real scenarios often involve more than one node. One reason for this, is that by XQuery

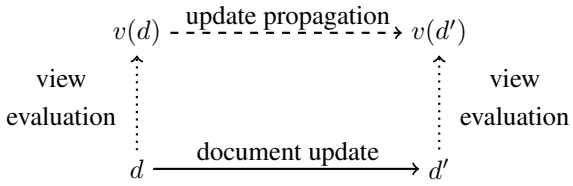


Fig. 1. The view maintenance problem

Update semantics, when node n is inserted in document d, all descendants of n become d nodes, thus adding n naturally leads to adding all its subtree. The same holds for deletions, i.e., removing n' from d automatically removes all the descendants of n' from d. Another reason is that updates can be performed within for-where XQuery expressions, again applying many node-level updates through a single statement. Repeatedly applying node-level update propagation procedures may become inefficient. Thus, we focus on *statement-level updates*, and study how to propagate in one step all the changes entailed by a given XQuery update statement to the affected view. This problem was studied in [Foster et al. 2008] which proposes an XQuery algebra-based approach for maintaining XQuery views. However, that approach is defined in the Galax algebra and is thus quite tied to the internals of that system. More information on related works is provided in Section 7.

In our work, we address the incremental maintenance of XML views in the presence of statement-level XML updates. Our view language corresponds to a core useful conjunctive XQuery subset. This language supports the child and descendant axis, value and branch predicates, and moreover allows returning data from more than one node, unlike the XPath dialects studied in [Björklund et al. 2009; Sawires et al. 2005; Sawires et al. 2006]. Our approach is designed to take advantage of advanced artifacts of current XML query processors, such as structural joins and smart identifiers [Xu et al. 2009]. Employing such efficient tools allows our algorithms to outperform node-level update propagation techniques in the frequent case where more than one node is added/removed at the same time. Moreover, our approach integrates smoothly in the process of updating the source document itself, by re-using some partial results of the update process. These features make it a good candidate to be integrated within a persistent XML database.

In summary, the originality of our approach lies in: *(i)* its algebraic, bulk-oriented character, relying on generic operators and *(ii)* the fact that we consider views returning data from multiple nodes. This paper is the extended version of our previous conference paper [Anonymous]. The new material brought in this version with respect to the conference paper is as follows. We have reported our deletion propagation algorithms which, for space reasons, were not included in the conference paper. We have added a new study on how to efficiently propagate sequences of updates, to a materialized view. Finally, our experimental analysis has been improved and new experiments have been added highlighting the performance of our algorithms.

This paper is organized as follows. Section 2 presents our model for documents, views, and updates. Section 3 provides algorithms for propagating insertions, whereas Section 4 studies deletions. Section 5 discusses the propagation of optimisation rules for sequences of updates. We study the performance of our algorithms in Section 6, compare our work in more detail with the state of the art in Section 7 and then conclude.

2. PRELIMINARIES

In this Section, we present our model for documents and node identifiers in Section 2.1, views in Section 2.2, and updates in Section 2.3.

2.1. XML documents and node identifiers

We view XML documents as ordered labeled trees, consisting of element, attribute and text nodes. Element nodes and attribute nodes have a label from a set of finite XML node labels, $\mathcal{L}$. Text nodes

have an associated string representing the value. Each node has a unique identifier (or ID, in short), which is given by a compact unique string in the corresponding encoding scheme. Among the many node ID schemes from the literature, we use structural IDs, and, more specifically, we consider the recently proposed compact dynamic Dewey IDs [Xu et al. 2009] because of their useful properties:

— they are structural, i.e., by comparing two nodes, it is possible to know whether one is a parent (or ancestor) of the other;
— from the ID of a node, one may extract the IDs and labels of all its ancestors;
— they do not require node relabeling in the presence of updates to the document;
— they can be encoded in a very compact fashion.

To better highlight the presence of IDs of a node, they are shown as a subscript of the node. Each structural ID is a sequence of steps, each step holding the label and the relative position of one ancestor of the node[1].

Figure 2 shows an example of such a representation for an arbitrary XML tree.

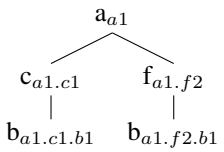


Fig. 2. Sample XML document

2.2. Views

We consider views expressed in a conjunctive XQuery dialect described in Figure 3. In this figure, $\mathcal{X}P$ stands for the XPath$^{\{/,//,*,[]\}}$ language; in the for clause, $absVar$ corresponds to an absolute variable declaration, which binds a variable named x_i to a path expression $p \in \mathcal{X}P$ to be evaluated starting from the root of some document available at the URI uri. The non-terminal $relVar$ allows binding a variable named x_i to a path expression $p \in \mathcal{X}P$ to be evaluated starting from the bindings of a previously introduced variable x_j. The optional where clause is a conjunction over a number of predicates, each of which compares the string value of a variable x_i with a constant c.

The return clause builds, for each tuple of bindings of the for variables, a new element labeled l, having some children labeled l_i ($l, l_i \in \mathcal{L}$). Within each such child, we allow one out of three possible information items related to the current binding of a variable x_k, declared in the for clause: (1) x_k denotes the full subtree rooted at the binding of x_k; (2) string(x_k) is the string value of the binding; (3) id(x_k) denotes the ID of the node to which x_k is bound.

Notice that the views may have multiple returned nodes, each one having a well-defined structure as specified in the return clause. There are important differences between the *subtree* rooted at an element (or, equivalently, its *content*), its *string value* and its *ID*. The content of x_i includes all (element, attribute, or text) descendants of x_i, whereas the string value is only a concatenation of x_i's text descendants [XPath 2.0 1999]. Therefore, string(x_i) is very likely smaller than x_i's content, but holds less information. Secondly, an XML ID does not encapsulate the content of the corresponding node. However, XML IDs enable joins which may stitch together tree patterns into larger ones. Our view dialect distinguishes between ID, value and content, and allows any subset of the three to be returned for any of the variables, resulting in significant flexibility.

Tree pattern representation for views For ease of explanation, we represent views using the following tree pattern dialect, denoted $\mathcal{P}$.

[1] Internally, of course, ID representation is much more compact.

1	q :=	(let $absVar$ return)? for ($absVar$,)? $relVar$ ($relVar$,)* (where $pred$ (and $pred$)*)? return ret	
3	$absVar$:=	x_i in doc(uri) $/p$	where $p \in \mathcal{X}P$
4	$relVar$:=	x_i in x_j $/p$	where x_j introduced before x_i
5	$pred$:=	string(x_i) = c	
6	ret :=	$\langle l \rangle$ $elem$* $\langle /l \rangle$	
7	$elem$:=	$\langle l_i \rangle \{$ (x_k/p | id(x_k) | string(x_k)) $\} \langle /l_i \rangle$	where $p \in \mathcal{X}P$

for $p in doc("confs")//confs//paper, $a in $p/affiliation
return ⟨result⟩ ⟨pid⟩{id($p)}⟨/pid⟩ ⟨aid⟩{id($a)}⟨/aid⟩ ⟨acont⟩{$a}⟨/acont⟩
⟨/result⟩

Fig. 3. Grammar for XML materialized views (top) and sample view (bottom).

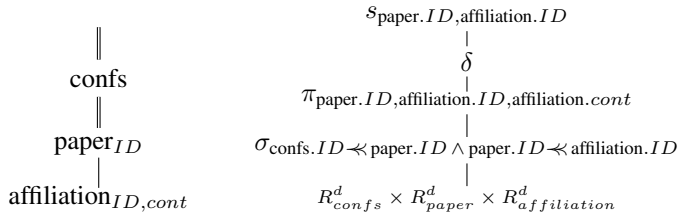


Fig. 4. Sample tree pattern (left) and corresponding algebraic semantics (right).

(1) Pattern nodes can carry the label of an XML element or attribute, or some word, respectively. A word is defined as a sequence of characters appearing either in a PCDATA node or in an attribute value, delimited by the usual separators (whitespace, tab, end of line). Each internal pattern node carries a label from a tag alphabet $A_l = \{a, b, c, \ldots\}$. Each leaf node carries a label from a word alphabet $A_w = \{\underline{a}, \underline{b}, \underline{c} \ldots\}$.
(2) Pattern edges correspond to parent-child or ancestor-descendant relationships between nodes.
(3) Each pattern node may be annotated with *stored attributes*, specifying the information items that the pattern stores out of each XML document node, that matches the pattern node. The $cont$ annotation indicates that the full (serialized) image matching the XML tree node is stored. The ID annotation indicates the Compact Dynamic Dewey ID [Xu et al. 2009] of the corresponding nodes. Storing IDs in views enables combining several views in order to answer a query [Tang et al. 2008; Manolescu and Zoupanos 2009]. Finally, the val annotation stands for the node's text value, obtained by concatenating all its text descendants in document order.
(4) Each node may be annotated with a predicate of the form $[val = \underline{c}]$ where $\underline{c} \in A_w$, restricting the XML nodes which match the pattern node, to those satisfying the predicate.

The translation of an XQuery view into an equivalent tree pattern is described (for a superset of the language considered here) in [Arion et al. 2006].

Algebraic tree pattern semantics View semantics can be defined in the customary way based on tree embeddings [Amer-Yahia et al. 2002]. For our purposes, we will rely on an equivalent semantics, introduced by means of an algebra. We present it here briefly, and point the reader to [Arion et al. 2005] for the detailed tree pattern semantics.

Given a document d and label $a \in \mathcal{L}$, we denote by R^d_a and call it the *virtual canonical relation of a in d*, the list of tuples of the form $(ID, val, cont)$ obtained from all the a-labeled nodes in d. The tuples in R^d_a are sorted in the order of appearance of the corresponding nodes in d. We denote by $\prec$ the *parent comparison operator*, which returns true if its left-hand argument is the ID of the parent of the node whose ID is the right-hand argument. Similarly, $\prec\!\!\prec$ is the *ancestor comparison*

operator. Observe that we only discuss a logical algebra here, and make no assumptions on how $\prec$ and $\prec\!\!\prec$ are actually implemented in a physical store.

Let $\mathcal{A}$ be the algebra consisting of the following operators: (1) the n-ary cartesian product $\times$; (2) selection, denoted σ_{pred}, where $pred$ is a conjunction of predicates of the form $a \odot \underline{c}$ or $a \odot b$, a and b are attribute names, $\underline{c}$ is some constant, and $\odot$ is a binary operator among $\{=, \prec, \prec\!\!\prec\}$; (3) projection, denoted π_{cols}; (4) duplicate elimination, denoted δ; (5) sort, denoted s_{cols}. For convenience, we also use joins, defined, as usual, as selections over $\times$.

For illustration, the algebraic semantics of the tree pattern at left in Figure 4 is the algebraic expression at right in the same figure. Considering the expression from the bottom up, there is an R_a atom per query node labeled a, and they are connected through $\times$ operators. The selection σ enforces (i) all value constraints on the nodes, and (ii) all structural $\prec$ or $\prec\!\!\prec$ relationships between query nodes. The projection retains the attributes projected by query nodes, e.g., paper.ID, affiliation.ID and affiliation.$cont$. After duplicate elimination (δ), we sort the tuples in the order dictated by the IDs of the bindings of all nodes.

Derivation count A final important note is needed on the view semantics. In keeping with the standard literature on view maintenance for relational and XML data [Gupta et al. 1993; Sawires et al. 2005], we associate with each tuple in a view a *derivation count*, which intuitively corresponds to the number of reasons why the tuple belongs to the view. In the semantics based on embeddings, the number of derivations of a tuple t corresponds to the number of distinct embeddings of the view in a document, that lead to the same view tuple t. In our algebraic semantics, the derivation count of t is the number of tuples in the input of the δ operator which lead to obtaining t.

We end by noting that the semantics (content) of a view v (corresponding to a tree pattern) on a document d, together with the derivation counts, can be computed in $O(|v| \times |d|)$, e.g., by an extension to the algorithm presented in [Chen et al. 2006].

2.3. Updates

We consider the following kinds of updates:

— delete q, where q is an XPath query from $\mathcal{XP}$;
— for $x in q insert xml into $x, where $q \in \mathcal{XP}$, and xml is a forest of XML trees. This generalizes to updates of the form for $x in q_1 insert q_2 into $x, where q_1 is any XQuery and q_2 is an XPath query, from $\mathcal{X}P$ as follows. First, we evaluate q_1 on the original document, and then we proceed as if we were inserting q_2 results as children. Of course q_2 may depend on $x, in which case different forests may be inserted under different nodes returned by q_1.
— the simpler form, insert xml into q with $q \in \mathcal{XP}$, as above is also supported, together with its more general variant, insert q_1 into q_2 where $q_1, q_2 \in \mathcal{XP}$.

3. PROPAGATING INSERTIONS

In this Section we start by outlining our overall approach for propagating insertions to views (Section 3.1), based on the algebraic semantic of tree patterns. This approach models an update as a union of algebraic terms to be added to, or deleted from the view. Section 3.2 provides a set of criteria which can be used to prune such terms, reducing the view propagation effort. In Section 3.3 we show how to detect the situations when an update may violate schema constraints. We then provide our algorithms for propagating updates: some helpful notions are introduced in Section 3.4, while the concrete algorithms are presented in Section 3.5 and 3.6.

3.1. Approach

Let v be a view whose nodes carry the names $a_1, a_2, \ldots, a_k \in \mathcal{L}$. Then, v can be written as:

$$v = e_v(\sigma_{a_1}(R_{a_1}) \bowtie \sigma_{a_2}(R_{a_2}) \bowtie \ldots \bowtie \sigma_{a_{k-1}}(R_{a_{k-1}}) \bowtie \sigma_{a_k}(R_{a_k}))$$

In the above, for each a_i, σ_{a_i} is a selection operator whose logical condition is: the value predicate attached to the view node labeled a_i, if such a predicate exists, or true otherwise. The algebraic

expression e_v includes the projections, sorts, and duplicate eliminations in the algebraic semantics of v. The joins $\bowtie$ correspond to the specific structural relationship predicates connecting the a_i nodes in the view v.

We designate the nodes added by u to d as *new* nodes. For any node label l, we term Δ_l^+ the ordered collection of tuples of the form $(n.ID, n.val, n.cont)$ obtained from all the nodes n added to the document by the update. The IDs of the new nodes are computed as a side-effect of the document update, whereas their values and contents can be extracted directly from the subtrees rooted at the nodes (recall that according to the XQuery update semantics, when a node n is added to d, all the subtree of n is added to d). Based on the Δ^+ relations, the impact of u on d can be expressed as follows: for each node label l occurring in v, replace R_l^d by $R_l^{d'} = R_l \cup \Delta_l^+$.

After the update, the content of the view v should thus become:

$$v' = v(d') = e_v(\sigma_{a_1}(R_{a_1} \cup \Delta_{a_1}^+) \bowtie \sigma_{a_2}(R_{a_2} \cup \Delta_{a_2}^+) \bowtie \ldots \bowtie \sigma_{a_{k-1}}(R_{a_{k-1}} \cup \Delta_{a_{k-1}}^+) \bowtie \sigma_{a_k}(R_{a_k} \cup \Delta_{a_k}^+))$$

Thus, v' can be obtained by evaluating the expression e_v over the result of a join expression. Our method to compute v' is to (i) compute e_v's input, i.e., the join expression and (ii) apply the unchanged algebraic expression e_v on the result. Given that (ii) is quite straightforward, we will ignore e_v in the sequel and focus on task (i), that is, efficiently and incrementally maintain the join expression.

Distributing the joins over unions in the expression above leads to a single union of 2^k *terms*, each of which is a join expression. One of which involves no Δ^+ relations, and corresponds to the original view v. Propagating u thus requires computing the remaining $2^k - 1$ union terms.

Example 3.1. Let d be a document and u_1 be an update that inserts in d the following XML snippet:

$$xml_1 = \langle a\rangle\langle b/\rangle\langle b\rangle\langle c/\rangle\langle/b\rangle\langle/a\rangle$$

Let a_1, b_1, b_2 and c_1 be the XML elements inserted in d by u_1. The Δ^+ relations corresponding to u_1 are:

Δ_a^+	Δ_b^+	Δ_c^+
$(a_1.id, a_1.val, a_1.cont)$	$(b_1.id, b_1.val, b_1.cont)$ $(b_2.id, b_2.val, b_2.cont)$	$(c_1.id, c_1.val, c_1.cont)$

Consider the view $v_1 = //a_{id}//b_{id}//c_{id}$. After u_1 is applied, v_1 should become:

$$\begin{array}{rl} v_1 & \cup (R_a \bowtie_{a \prec\!\!\prec b} R_b \bowtie_{b \prec\!\!\prec c} \Delta_c^+) \cup \\ (R_a \bowtie_{a \prec\!\!\prec b} \Delta_b^+ \bowtie_{b \prec\!\!\prec c} R_c) & \cup (R_a \bowtie_{a \prec\!\!\prec b} \Delta_b^+ \bowtie_{b \prec\!\!\prec c} \Delta_c^+) \cup \\ (\Delta_a^+ \bowtie_{a \prec\!\!\prec b} R_b \bowtie_{b \prec\!\!\prec c} R_c) & \cup (\Delta_a^+ \bowtie_{a \prec\!\!\prec b} R_b \bowtie_{b \prec\!\!\prec c} \Delta_c^+) \cup \\ (\Delta_a^+ \bowtie_{a \prec\!\!\prec b} \Delta_b^+ \bowtie_{b \prec\!\!\prec c} R_c) & \cup (\Delta_a^+ \bowtie_{a \prec\!\!\prec b} \Delta_b^+ \bowtie_{b \prec\!\!\prec c} \Delta_c^+) \end{array}$$

For brevity, in the sequel, we will omit the join predicates (which are always those of the view) from the union terms. Thus, the expected content of v_1 after the insertion in Example 3.1 can be written as:

$$v_1 \cup R_a R_b \Delta_c^+ \cup R_a \Delta_b^+ R_c \cup R_a \Delta_b^+ \Delta_c^+ \cup \Delta_a^+ R_b R_c \cup \Delta_a^+ R_b \Delta_c^+ \cup \Delta_a^+ \Delta_b^+ R_c \cup \Delta_a^+ \Delta_b^+ \Delta_c^+$$

In the following, we study practical algorithms for computing the $2^k - 1$ terms whose results need to be added to v in order to make it reflect the insertion.

3.2. Term pruning

Several observations lead us to infer when some of the union terms are guaranteed to have empty results. Such union terms are *pruned*, that is, their evaluation is not necessary in order to propagate the insertion to the view. Pruning significantly reduces the update propagation effort and can be generalized by the following propositions.

Pruning by the update semantics The semantics of XQuery Update allows determining that some terms will always have empty results. Intuitively, this is because XQuery updates allow adding new children to existing nodes, but not new parents, as the following example illustrates.

Example 3.2. Consider the insertion u_1 from Example 3.1. A newly added a node cannot have as child a b node which belonged to d before u_1 was applied. Thus, $\Delta_a^+ R_b$ is empty, therefore the terms $\Delta_a^+ R_b R_c$ and $\Delta_a^+ R_b \Delta_c^+$ to be added to v_1 in order to maintain it are guaranteed to produce an empty result. Similarly, no b element in Δ_b^+ can have descendants in R_c, therefore $\Delta_b^+ c$ is also empty, and so are $R_a R_b \Delta_c^+$ and $\Delta_a^+ R_b \Delta_c^+$. Thus, to compute v_1', it suffices to add to v the results of evaluating the terms:

$$R_a R_b \Delta_c^+ \cup R_a \Delta_b^+ \Delta_c^+ \cup \Delta_a^+ \Delta_b^+ \Delta_c^+ \qquad (*)$$

This is generalized by the following proposition:

PROPOSITION 3.3. *Let v be a view of k nodes, and n_1, n_2 be v nodes such that n_2 is a ($/$ or $//$) child of n_1. Let R_{n_1}, respectively, R_{n_2} be the atoms corresponding to n_1, respectively, n_2 in the algebraic semantics of v, i.e., $R_{n_1} \bowtie R_{n_2}$ is a sub-expression of v. Let u be an arbitrary insertion, and t be one of the $2^k - 1$ terms to be added to v in order to propagate the effect of u. If t contains as a sub-expression $\Delta_{n_1}^+ R_{n_2}$, then t produces an empty result.*

Observe that Proposition 3.3 *does not depend on the insertion u*. Therefore, in the following, we only focus on the terms which survive this pruning.

Inserted data-driven pruning Inspecting the XML fragments may allow further pruning, as illustrated by the following example:

Example 3.4. Consider the view v_1 from Example 3.1 and the insertion u_2 which adds the following XML snippet:

$$xml_2 = \langle a \rangle \langle b/ \rangle \langle b/ \rangle \langle /a \rangle$$

The difference with respect to Example 3.1 is that xml_2 does not include a c element, i.e., $\Delta_c^+ = \emptyset$. This entails that all the terms of the expression $(*)$ in Example 3.2 are empty and thus, v_1 is not affected by u_2.

Value predicates may also impact update propagation, as the following example shows:

Example 3.5. Consider the view $v_2 = //a_{[val=5]}//b_{id}$ and the insertion u_3 adding the following XML snippet:

$$xml_3 = \langle a \rangle 3 \langle b/ \rangle \langle b/ \rangle \langle /a \rangle$$

In this case, $\Delta_a^+ \neq \emptyset$ and $\Delta_b^+ \neq \emptyset$, however $\sigma_b(\Delta_b^+) = \emptyset$ because the new a element does not satisfy the view predicate $[val = 5]$. Thus, $R_a \Delta_b^+$ and $\Delta_a^+ \Delta_b^+$, which both involve $\sigma_b(\Delta_b^+)$, are empty. Since by Proposition 3.3, term $\Delta_a^+ b$ is also empty, v_2 is unaffected by u_3.

This generalizes to the following simple observation:

PROPOSITION 3.6. *Let u be an insertion adding the trees $t_1, t_2, \ldots, t_k$ to d, and v be a view. If a node n of v is not matched in any of the trees $t_1, t_2, \ldots, t_k$, all union terms involving Δ_n^+ are empty.*

Inserted ID-driven pruning A third pruning criteria reasons on the *label paths leading to the insertion points*, encoded in their Compact Dynamic Dewey IDs [Xu et al. 2009]:.

Example 3.7. Consider the view v_1 from Example 3.1 and the insertion u_4, adding the following XML snippet:

$$xml_4 = \langle b \rangle \langle c/ \rangle \langle /b \rangle$$

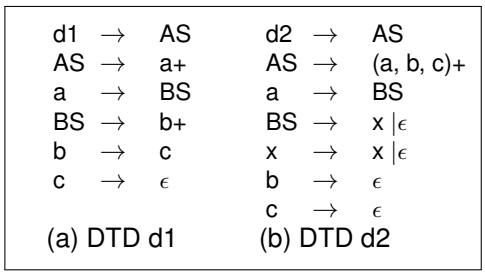


Fig. 5. **Sample DTDs, expressed as CFGs.**

as a child of a node a, whose ID is $a.id$. This ID encodes the labels of all the nodes on the path from a to the root [Xu et al. 2009]. Assume that we inspect $a.id$ and find that no ancestor labeled b appears above the a node. Then, the new (inserted) c node has only one b ancestor, namely the inserted (new) b node. Thus, the term $R_a R_b \Delta_c^+$ in $(*)$ is empty.

In this example, moreover, since $\Delta_a^+ = \emptyset$, the term $\Delta_a^+ \Delta_b^+ \Delta_c^+$ in $(*)$ is also empty. Thus, the only term we need to compute to update v_1 after inserting xml_4 is: $R_a \Delta_b^+ \Delta_c^+$.

This generalizes as follows:

PROPOSITION 3.8. *Let u be an insertion adding children to the nodes $p_1, p_2, \ldots, p_k$ in d. Let v be a view, and n_1, n_2 be v nodes such that n_1 is an ancestor of n_2 in v.*

If for each $i = 1, 2, \ldots, k$, p_i is not labeled n_1 and has no ancestor labeled n_1, then all union terms containing $R_{n_1} \Delta_{n_2}^+$ are empty.

3.3. Detecting schema violations

The standard W3C model for XML queries and updates does not require schema information describing the documents. However, when a schema is available, it can be used to reason about the impact of an update, directly on the document (will the document still be valid according to the schema, after a given update?) or on the view (is this view impacted by this update?). The latter problem is known as view-update independence, and has been thoroughly considered elsewhere [Balmin et al. 2004b; Benedikt and Cheney 2009]. In this Section, we briefly illustrate how our Δ^+ tables could be used at runtime, to solve the former problem: decide whether an update could violate the document schema. When presented at runtime with the information that the update may violate the schema, the user may chose whether to proceed or reformulate the update.

For our purpose, we consider that documents are characterized by DTDs expressed as extended context-free grammars (CFGs), where the right-hand side of each rule is a regular expression over an alphabet of terminal and non-terminal symbols. For instance, Figure 5 depicts two DTDs. In this Figure, a, b, c, d1, d2 and x are terminal symbols and AS and BS are non-terminal ones. DTD d1 (a) has mandatory edges, while DTD d2 (b) features concatenation, disjunction and recursion.

Example 3.9. Consider the view v_1 from Example 3.1 and an insertion u_5, adding the following XML snippet:

$$xml_5 = \langle a \rangle \langle b \rangle \langle / b \rangle \langle / a \rangle$$

Applying the update would make the document invalid with respect to the DTD in Figure 5(a), since a c element is missing under b. More generally, from the DTD d1, one can derive that the following statement must hold for any newly inserted XML tree:

$$\Delta_c^+ = \emptyset \Rightarrow \Delta_b^+ = \emptyset$$

Since this does not hold on the update u_5, we reject it due to its attempted schema violation.

The same consideration applies to Figure 5(b), in which a $d2$ element must have as children the concatenation of a, b and c. Therefore, any insertion of an a element under the root $d2$ must occur with b and c elements.

Example 3.10. The DTD in Figure 5(b) implies that the following statement must hold on any XML forest inserted under a given node:

$$\Delta_a^+ \neq \emptyset \Rightarrow (\Delta_b^+ \neq \emptyset \wedge \Delta_c^+ \neq \emptyset)$$

More generally, from the DTD rules, one can infer a set of constraints on the Δ^+ tables, and check them before applying the update.

3.4. Helper functions and operators

Let u be an update (insertion or deletion), and $pul(u)$ be the **pending update list** [XQuery Update Facility 1.0 2009] resulting from u. Thus:

— if u is an insertion, $pul(u) = \{(n_1, t_1), (n_2, t_2), \ldots, (n_k, t_k)\}$, a list of pairs consisting of an XML element n_i *target of the update*, and a subtree t_i to be copied as a child of n_i.
— if u is a deletion, $pul(u) = \{n_1, n_2, \ldots, n_k\}$ is the list of the nodes to be removed.

We assume available:

— **compute-pul(**u**)** is a function which from an update u, computes its pending update list $pul(u)$.
— **apply-insert(**n**,**t**)** is a side-effect function which, given a node n and a tree t, copies t into a new tree t', inserts t' as a new child of n and returns t'. Importantly for us, *the tree t' also includes the IDs assigned to the copied t nodes in their new context (in d).*
— **extr-pattern(**p**,** t**)** is a function which, given a tree pattern p and an XML tree t, evaluates p on t and returns the corresponding set of tuples.
— **physical operators** including for instance structural joins [Al-Khalifa et al. 2002]. Based on Compact Dynamic Dewey IDs [Xu et al. 2009], we also use Path Filter, for checking whether a node having a specific ID is on a path satisfying a specific condition, and Path Navigate for obtaining, from the IDs of nodes, the IDs of their parents.

3.5. Propagating insertions by adding tuples

Our first update propagation algorithm considers the case when an insertion to the XML document either leads to new tuples being added to the view, or does not affect the view. The cases when the insert leads to *modifying* view tuples will be addressed in Section 3.6.

Algorithm 1 outlines the propagation procedure. The first step (developing the union terms corresponding to v) is performed when v is created, and it is independent of the updates. The computation of the Δ^+ tables will be detailed shortly. The core of the complexity in Algorithm 1 lies in the computation of the union terms which are not pruned by our criteria.

ALGORITHM 1: Propagate Insert by New Tuples **(PINT)**

Input: view v, insert update u
Output: updated view v' to reflect u
Develop the $2^k - 1$ union terms to be added to v in case of insertions, and prune them based on XQuery update semantics (Proposition 3.3)
Compute the Δ^+ tables corresponding to u (call Algorithm **CD+**(v, u))
Further prune terms based on the Δ^+ tables (Propositions 3.6 and 3.8)
Evaluate the remaining terms and add their results to v (call Algorithm **ET-INS**(u, v, sc))
If needed, update auxiliary structures

ALGORITHM 2: Compute Δ^+ tables (**CD+**)

Input: update u, view v
Output: Δ^+ tables
$(n_1, t_1), \ldots, (n_k, t_k) \leftarrow$ **compute-pul**(u)
for $n \in v$ *labeled* l **do**
  let p_l be the pattern $//l_{ID,val,cont}$
  $\Delta_l^+ \leftarrow \bigcup_{1 \leq i \leq k}$(**extr-pattern**($p_l$, t_i))

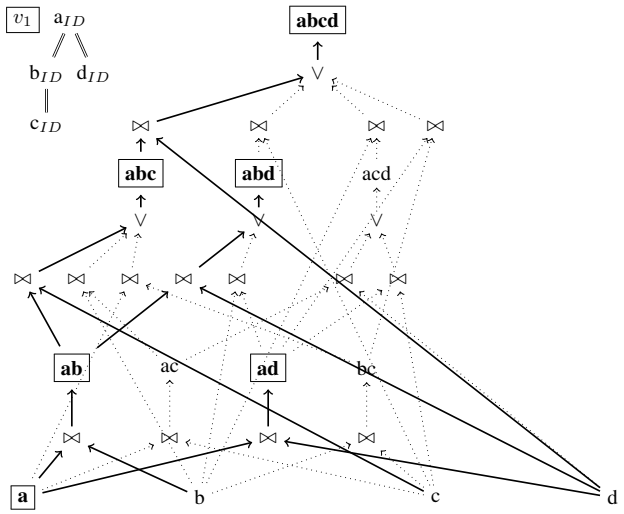


Fig. 6. Sub-pattern lattice and snowcaps for the view v_1.

Computing Δ^+ relations Given an insertion u on the document d and the view v, Algorithm 2 computes the Δ^+ relations. It relies on the functions **compute-pul** and **extr-pattern** presented in Section 3.4 to compute the update list, and then to extract Δ^+ relations out of the pending updates.

Term evaluation based on auxiliary lattice To maintain the view v through the union terms which survived pruning, we rely on a collection of auxiliary data structures, organized in a *view lattice*, which is an AND-OR graph specialized for our problem. Figure 6 depicts the lattice corresponding to the view $//a_{ID}[//b_{ID}//c_{ID}]//d_{ID}$. Formally, the lattice of v is a DAG with three kinds of nodes: (i) *pattern-labeled nodes*. One node is labeled by the pattern v. Each other pattern-labeled node is labeled by a distinct sub-tree pattern of v. For readability, in Figure 6 and in the sequel, a pattern labeled node is designated simply by the set of the pattern's node labels. (ii) A set of or-nodes labeled $\vee$; (iii) A set of join nodes labeled $\bowtie$.

Lattice edges trace possible ways of computing a pattern-labeled node based on those below it. For instance, a join ($\bowtie$) allows computing the node labeled ab, corresponding to $//a_{ID}//b_{ID}$, out of the nodes labeled a and b respectively. When the sub-pattern corresponding to a lattice node can be computed in several ways out of the lower nodes, this is modeled by the or ($\vee$) node which alone points to the target lattice node. In Figure 6, the sub-pattern abc can be computed in three distinct ways.

Materializing (and maintaining) all pattern-labeled lattice node suffices to maintain v after an insertion. Indeed, for each union term t to be added to v:

— Let t_R be the $\bowtie$ sub-expression(s) of t containing only R_a occurrences. Then, t_R corresponds exactly to some materialized lattice node(s).

— Let t_{Δ^+} be the $\bowtie$ sub-expression(s) of t containing only Δ^+ occurrences. Then, t_{Δ^+} is easily computed based on the newly inserted data.

The lattice provides a blueprint for solving the problem of maintaining v, based on the "smaller" problems of maintaining the sub-patterns of v. The maintenance of the lattice leaves is trivial: replace R_a by $R_a \cup \Delta_a^+$. Thus, maintaining all the expressions corresponding to lattice nodes, in a bottom-up fashion, is guaranteed to maintain v. However, materializing and maintaining all lattice nodes is likely to be very expensive in terms of space and time. Fortunately, we can focus only on a subset of these nodes:

Definition 3.11 (*Snowcap*). Let v be a tree pattern. We term *snowcap of* v, any non-empty subtree t of v such that: for each node $n \in v$ that also appears in t, the parent of n in v also appears in t.

For instance, in Figure 6, the boxed nodes depict snowcaps. Intuitively, a snowcap copies the root of the pattern and then goes down to some length on all paths, *only including a node n if it includes its parent*. This mimics the way mountains are covered by snow from the top downward, thus the name. We can now state:

PROPOSITION 3.12. *Let v be a view, u an insertion and t a term resulting from u. The term t survives our first pruning (Proposition 3.3) if and only if t_R (the sub-expression of t which does not contain Δ^+ symbols) is the algebraic semantics of a pattern v_{t_R}, which is a snowcap in v's lattice.*

"If" direction: we show that if t_R corresponds to a snowcap in v's lattice, then t is not eliminated by Proposition 3.3. Since v_{t_R} is a node in v's lattice, it corresponds to a subset N_{t_R} of v's nodes. The term t can be written as a join between t_R and the Δ^+ tables for all the v nodes that are not in N_{t_R}, where the join predicates are derived from the structure of v. Let n_1 be a v node and n_2 a $/$ or $//$-child of n_1. Since v_{t_R} is a snowcap, exactly one of the following must hold:

— n_1 and n_2 are both in v_{t_R}, therefore R_{n_1} and R_{n_2} are both in t_R and t. In this case, n_1 and n_2 do not lead to t being pruned by Proposition 3.3.
— n_1 is in v_{t_R} thus R_{n_1} is in t_R (and t), while n_2 is not in v_{t_R} and thus $\Delta_{n_2}^+$ appears in t. The t sub-expression $R_{n_1}\Delta_{n_2}^+$ does not trigger the pruning of Proposition 3.3, either.
— neither n_1 nor n_2 are in v_{t_R}, thus they do not appear in t_R and t features $\Delta_{n_1}^+\Delta_{n_2}^+$ as a sub-expression, which again cannot trigger the condition of Proposition 3.3.

"Only if" direction: we show that if t survives the pruning of Proposition 3.3, then t_R corresponds to a snowcap in v's lattice. If t includes no Δ^+ table, then $t_R = t$ coincides with the lattice topmost node, which is indeed a snowcap. Now assume that t includes at least one Δ^+ table, corresponding to the view node n. Moreover, since t was not pruned by Proposition 3.3, it follows that for any node n' immediately under n in v, t also contains the Δ^+ table corresponding to n'; repeating this reasoning shows that for t' contains $\Delta_{n'}^+$ for any descendant n' of n. One of the two cases below must hold:

(1) For any ancestor m of n in v, t includes Δ_m^+. Thus, t includes the Δ^+ corresponding to v's root, therefore t includes Δ^+ tables for all v nodes, thus $t_R = \emptyset$ which verifies our conclusion.
(2) There exists a lowermost ancestor m of n such that t does not contain Δ_m^+ (equivalently, such that R_m appears in t_R). Then it is easy to see that for any ancestor m' of m, $R_{m'}$ appears in t_R; in other words, all nodes from m upward to v's root must appear in t_R.

Generalizing the above reasoning to all v nodes n such that Δ_n^+ appears in t, we obtain that for any node m in v_{t_R}, all ancestors of m are in v_{t_R}. In other words, v_{t_R} is a snowcap. □

As an example of application of this Proposition, consider the view v_1 in Figure 6. For an insertion u to add tuples to v_1, one or several of the following cases must hold: (i) u adds a d child to an element on the path $//a//b//c$. The impact of this addition on v is obtained by joining the snowcap abc with Δ_d^+; (ii) u adds a c child to an element matching $//a//b$, and this c child has at least

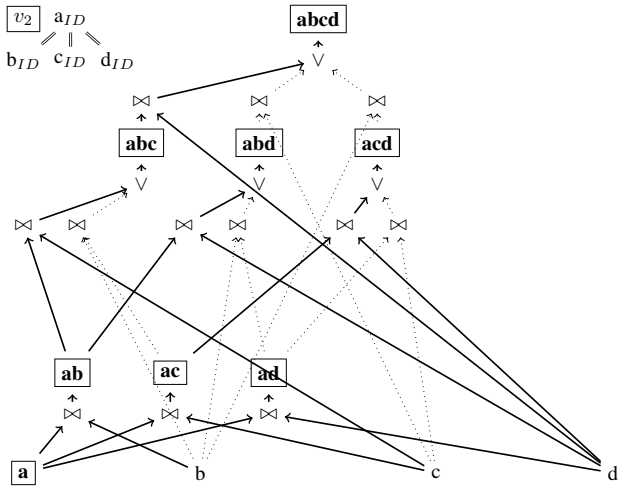


Fig. 7. Sub-pattern lattice and snowcaps for the view v_2.

one d descendant (join the snowcap ab with $\Delta_c^+\Delta_d^+$); (iii) u adds a b child to an element matching $//a$, and this b child has at least one $//c//d$ descendant (join the snowcap a with $\Delta_b^+\Delta_c^+\Delta_d^+$); or (iv) u adds matches to the full $//a[//b//c]//d$ view path, and to reflect such additions, no auxiliary structure (lattice node) is needed.

This analysis of the possible ways in which an insertion could add tuples to a view demonstrates that in such an example, the snowcap pattern-labeled lattice nodes are necessary and sufficient to maintain the view. We generalize this into the following proposition:

PROPOSITION 3.13. *Each snowcap can be maintained based only on other snowcaps, the lattice leaves and the Δ^+ relations extracted during an update.*

PROOF. The proof is by induction on the number of nodes k of the snowcap tree pattern. If $k = 1$, the snowcap is a leaf, and it can be maintained by adding to it the corresponding Δ^+ tuples. If $k > 1$, by the induction hypothesis, any snowcap of $k - 1$ nodes can be maintained based on smaller snowcaps. Moreover, there exists a snowcap s_{k-1} such that (i) s_{k-1} has all the view nodes appearing in s_k but one, (ii) there is a $\bowtie$ node in the lattice pointing to the $\vee$ node which points to s_k, and such that the s_{k-1} node points to the $\bowtie$. This means that s_k can be incrementally maintained by joining the tuples added to s_{k-1} (as part of its own incremental maintenance) with those from the leaf node that is in s_k but not in s_{k-1}. □

Observe that since v is a snowcap itself, it trivially follows that maintaining the snowcaps is sufficient to maintain v.

Snowcaps are sufficient to maintain a view. However, there may be more snowcaps than v maintenance requires, as illustrated in Figure 7. One could maintain **abcd** by maintaining **abc**, **ab** and **a**; the other snowcap terms are not needed. The proof of proposition 3.13 actually shows that each snowcap can be maintained by joining *one* smaller snowcap with a leaf etc. The most efficient node combination clearly depends on the data set statistics, governing the size of each sub-pattern in the lattice, the size of the Δ^+ tables corresponding to the various view nodes (which can be seen as reflecting the nature of the update, since some elements may be added in greater numbers than others), the join order, join processing costs etc. In our experiments, we have compared for illustration two simple alternatives: maintaining a minimal set of snowcaps and leaves, versus using only the lattice leaves and re-computing the internal joins on the fly. The results are provided in Section 6.7.

ALGORITHM 3: Evaluate terms resulting from insert (**ET-INS**)

Input: update u, view v, materialized snowcaps sc
Output: updated view v to reflect u; updated materialized snowcaps sc
Evaluate Δ^+ tables (call Algorithm **CD+**(u, d))
$\Delta_v^+ \leftarrow \emptyset$ (tuples to be possibly added to v)
foreach *term* t *surviving pruning* **do**
  Evaluate t_{Δ^+} by structural joins over the Δ^+ tables
  Add to Δ_v^+ the result of joining t_R (snowcap materialized in the lattice) and t_{Δ^+}
foreach *tuple* $t_\Delta \in \Delta_v^+$ **do**
  if $t_\Delta \in v$ **then**
    increase the derivation count of t_Δ
  else
    add t_Δ to v with a derivation count of 1

Algorithm 3 (**ET-INS**) outlines the evaluation of non-pruned terms resulting from insertions. In the first for loop, Algorithm **ET-INS** employs structural joins, taking advantage of efficient evaluation techniques within the XML query engine.

Optimal choice of snowcaps A question arising from the above discussion is: which snowcaps to use in order to maintain the views most efficiently? The choice must clearly be guided by estimating the expected performance of the maintenance procedure. In turn, this performance is impacted by several factors, which include:

— The characteristics (structure, size, etc.) of the *existing* data. Such statistics are obviously needed for the basic functioning of the XML database in itself (even in the absence of any materialized views). They can be computed, e.g., under the form of XSKETCH summaries [Polyzotis and Garofalakis 2006], and employed by a cost-based query optimizer taking into account data access costs, costs for the available join operators, join orders, etc. Some frameworks for cost-based XML query optimizers were detailed, e.g., in [Wu et al. 2003; Georgiadis et al. 2009].
— The characteristics of the update stream: what are the types of updates received by the database, and what is their impact? For instance, we may know that ⟨bookLoan⟩ elements are only ever added; also, ⟨bookLoan⟩ elements are modified to add to each of them exactly one ⟨loanEnds⟩ child, or alternatively, a ⟨loanRenewed⟩ child; no element is ever deleted, etc. Such update profiles may be obtained by analyzing the application code (including the corresponding XQuery Update statements), or they can be extracted from execution logs etc. As a matter of fact, such information is routinely gathered as part of the database server *workload* [Yu et al. 1992]. Indeed, information about the update batch to refresh the database along with online updates is recorded and accounted for in a typical workload production. From this update profile, we can derive estimations of how often each view will need to be updated, and even more precisely, how often will it need to be updated to reflect the addition of new ⟨a⟩ elements.

Based on (i) the expected rate of changes (e.g., arrival of new ⟨a⟩ elements), (ii) the algebraic expression reflecting each snowcap in the lattice of v, that includes ⟨a⟩ nodes and (iii) estimations of the cost of the respective operations of the form $\Delta_a^+ \bowtie R_b \bowtie \ldots$, one can choose the snowcaps most suitable for maintaining a given view v. Finally, it is worth noting that such optimizations may be carried in a more global fashion. In a context where several views are materialized and some snowcaps may be shared, it makes sense to sum up the respective maintenance costs and pick a set of snowcaps sufficient for maintaining all the views with an overall optimal performance.

The discussion above shows that the optimal choice of snowcaps is a cost-based optimization decision, which can be made within the XML database relying on information and primitives it already has. For space and scope reasons, we do not develop this further in the present paper, but

delegate it to future work. In our experiments (Section 6), we relied on simple choices of snowcaps that are sufficient for our observations.

3.6. View tuple modification

An insertion may lead to modifying existing tuples of a view v. This occurs when the insertion changes the value or the content of an XML node n, whose value (respectively, content) is stored in v. In turn, the value and content of n may also change as a consequence of adding or modifying a descendant of n.

Example 3.14. Consider the view $/a_{ID}/b_{ID}//c_{ID,Cont}$ and an insertion u adding the XML snippet:

$$\langle extra \rangle \text{ some value } \langle /extra \rangle$$

into $//d//c$. In this case, no Δ^+ relation affects the view, thus no new tuples need to be added. However, the insertion u may lead to modifying some of the $c.cont$ values stored by the view, if the intersection of $/a/b//c$ and $//d//c$ is not empty.

In the following, we present an algorithm that addresses this case. The algorithm considers all XML nodes for which the view stores content or value, verifies whether those nodes are affected by the update, and if this is the case, updates their value and/or content.

ALGORITHM 4: Propagate Insert by Modifying Tuples **(PIMT)**

Input: insert update u, view v
Output: updated view v' to reflect u
$ut = [(n_1, t_1), \ldots, (n_k, t_k)] \leftarrow$ **compute-pul**(u);
$cvn \leftarrow \{n \in v, n \text{ annotated with } cont \text{ or } val\}$;
foreach *tuple* $t \in v$ **do**
  foreach *tuple* $(n_i, t_i) \in ut$ *and* $t.n \in cvn$ **do**
    if $t.n = n_i$ *or* $t.n \prec\!\!\prec n_i$ **then**
      Update $t.n.cont$ (respectively, $t.n.val$) to reflect the insertion of t_i

Algorithm 4 (**PIMT**) starts by computing the pending update lists. It singles out all the content- or value-annotated view nodes, in the node set cvn. The algorithm then checks, for each view tuple t, whether and how each of the t attributes corresponding to the content or value of a cvn node must change. To that purpose, Algorithm 4 requires that for all cvn nodes, i.e., for all those view nodes for which $cont$ or val is stored, that element IDs must also be stored. Based on the IDs, we check whether the node providing the $cont$ attribute in tuple t is the same as, or an ancestor of the modified node n_i. If this is the case, then the insertion of t_i has to be propagated to the t attribute corresponding to $n.cont$ (respectively, $n.val$). It is easy to see that if cvn is empty, insertions cannot modify view tuples (but only add to the view).

If cvn is of size 1 (a single node of v stores val or $cont$), Algorithm **PIMT** can be implemented by a single efficient structural join (extended to check ancestor-descendant *or* equality relationships) between v and the pending update list. In the view tuples that join with the pending update list, the $cont$ and/or val attributes must be changed.

If cvn contains more nodes, Algorithm 4 must compare several ID attributes from each view tuple t against the pending update list, and a nested loops join is needed.

We end by noting that in practice an insertion may lead to tuples being both added and modified. Therefore, in practice we run a combined **PINT/MT** algorithm, which computes the pending update list only once and then applies the steps of both **PINT** and **PIMT**, based on the lattice. The lattice is also obviously updated only once.

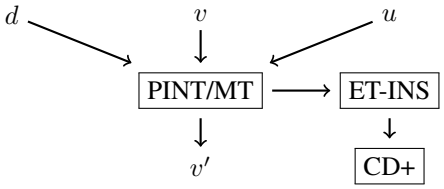


Fig. 8. Insert propagation outline.

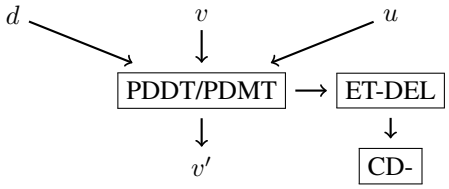


Fig. 9. Delete propagation outline.

Symbol	Description
d	Document
v	View
v'	Updated View
u	Update
PINT	Propagate Insert by New Tuples (Algorithm 1)
ET-INS	Evaluate Terms Resulting from Insert (Algorithm 3)
CD+	Compute Δ^+ Tables (Algorithm 2)
PIMT	Propagate Insert by Modifying Tuples (Algorithm 4)
PDDT	Propagate Delete by Deleting Tuples (Algorithm 5)
ET-DEL	Evaluate Terms Resulting from Delete (not shown, similar to ET-INS)
CD-	Compute Δ^- Tables (not shown, similar to CD+)
PDMT	Propagate Delete by Modifying Tuples (not shown, similar to Algorithms 5 and 4)

Fig. 10. Acronyms used in Figure 8 and Figure 9.

The main steps of the insert propagation procedure are shown in Figure 8. The symbols used in the figure appear together with their explanation in Figure 10.

PROPOSITION 3.15 (COMPLEXITY OF ALGORITHMS 1 AND 4). *Let v be a view of k nodes and u be an insertion resulting in a pending update list PUL. Let R be the largest relations among the leaf nodes in v's lattice. The worst-case complexity of Algorithm 1 (PINT) is $\mathcal{O}(2^k \times k \times max(|PUL|, |R|))$. Moreover, the complexity of Algorithm 4 (PIMT) is $\mathcal{O}(|v| \times |PUL| \times |cvn|)$, where cvn is the set of content-returning nodes of v.*

The complexity of Algorithm 1 (PINT) is dominated by the evaluation of terms in Algorithm 3 (ET-INS) (called by Algorithm 1). The upper bound is obtained by assuming (pessimistically) that no term is pruned; also, we assume (pessimistically) that no intermediary node lattice is materialized, thus joins have to be re-computed from the lattice leaves and the Δ^+ tables. Observe that we do not account for the effort to compute the Δ^+ tables in the complexity of PINT, since we assume their computation is part of the update process itself. Moreover, the size of a Δ^+ table is bound by $|PUL|$. The inputs to the k-way join of each term are either a Δ^+ table or a leaf in the lattice (whose size is bound by $|R|$). Finally, we assume that efficient join algorithms such as the holistic twig joins allow evaluating a term in time proportional to the cumulated size of its inputs.

4. PROPAGATING DELETIONS

We now turn to the propagation of deletions, which may lead to deleting or modifying view tuples. The following example illustrates a simple deletion scenario.

Example 4.1. Consider the view $//a_{ID}//b_{ID}$ and the XML document d shown in Figure 11, where the ID of each node is shown as a subscript. Consider an update u_1 deleting $//c//b$. When evaluated on the document, this results in the node whose ID is $a1.c1.b1$, which must be deleted from the document. Therefore, the tuple $(a1, a1.c1.b1)$ must be removed from the view.

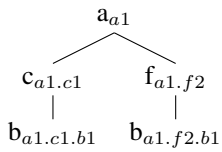


Fig. 11. Sample XML document

In the following, Section 4.1 outlines the basis of our algebraic deletion propagation approach, namely algebraic delete expressions. Section 4.2 provides term pruning criteria to simplify the deletion expression based on the semantics of XML and XML updates, while Section 4.3 does the same based on the actual data removed by the deletions. Finally, Section 4.4 provides our update propagation algorithms due to deletions.

4.1. Update propagation expression

In the context of a deletion, and for a given view node label a, we term Δ^-_a the ordered collection of tuples of the form $(n.id)$ obtained from all nodes n to be deleted, which are labeled a. Due to a deletion, a view of the form:

$$R_{a_1} \bowtie R_{a_2} \bowtie \dots R_{a_k}$$

needs to be transformed after the possible deletions of the elements in $\Delta^-_{a_1}$, $\Delta^-_{a_2}$, …, $\Delta^-_{a_k}$ into:

$$(R_{a_1} \setminus \Delta^-_{a_1}) \bowtie (R_{a_2} \setminus \Delta^-_{a_2}) \bowtie \dots \bowtie (R_{a_k} \setminus \Delta^-_{a_k})$$

One can expand this based on the known properties of the relational $\bowtie$ (join) and $\setminus$ (set difference) operators, eliminating the parentheses and reaching an expression of the form:

$$R_{a_1}R_{a_2}\dots R_{a_k} \setminus \Delta^-_{a_1}R_{a_2}R_{a_3}\dots R_{a_k} \setminus R_{a_1}\Delta^-_{a_2}R_3\dots R_k \setminus \dots \setminus R_{a_1}R_{a_2}R_{a_3}\dots\Delta^-_{a_k}$$
$$\cup\, \Delta^-_{a_1}\Delta^-_{a_2}R_3\dots R_{a_k} \cup R_{a_1}\Delta^-_{a_2}\Delta^-_{a_3}\dots R_{a_k} \dots \Delta^-_{a_1}\Delta^-_{a_2}\Delta^-_{a_3}\dots\Delta^-_{a_k}$$

The expanded expression above has 2^k terms. The first term is v, whereas all the others involve between 1 and k Δ^- tables. We call it the *(expanded) deletion expression of* v. Propagating an update u, which gives concrete values to the Δ^- tables, to the view v amounts to computing this expression, which in turns requires the evaluation of all its terms.

4.2. Update-independent term pruning

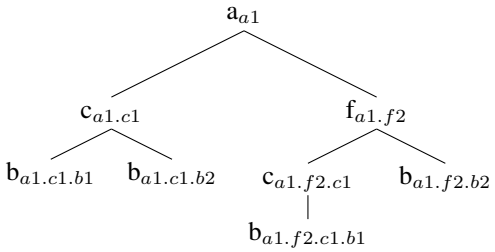
To simplify the processing of updates, we identify several criteria for term pruning.

PROPOSITION 4.2. *Let v be a view and n_1, n_2 be two nodes in v, such that node n_2 is a / or // child of n_1 in v. Let t be a term in a deletion expression, such that $\Delta^-_{n_1}R_{n_2}$ is a sub-expression of t. Then, t has empty results.*

This Proposition holds simply because in the XQuery delete model, when one removes a node matching n_1, implicitly all descendants of this node (including those matching the node n_2) are also removed. Just like its counterpart for insertions in Proposition 3.3, Proposition 4.2 is independent of the update, thus it can be applied directly (and very quickly) based on the view syntax alone.

PROPOSITION 4.3. *Let v be a view and t be a term in its deletion expression, including $k > 0$ relations of the form Δ^-. (i) If k is odd, the operand before t is $\setminus$, that is, the tuples of t must be removed from v, whereas if k is even, the operand before t is $\cup$, that is, the deletion expression requires adding these tuples to reflect the deletion; (ii) if k is even, one can ignore t from the deletion expression.*

PROOF. Claim (i) follows directly from when one distributes the joins over the set difference in the deletion expression. For what concerns claim (ii), an even and positive k must be at least 2. This ensures that in the deletion expression there exist k terms $t_1, t_2, \dots, t_k$ such that for each $1 \leq i \leq k$,



$num.$	a_{ID}	c_{ID}	b_{ID}
1	$a1$	$a1.c1$	$a1.c1.b1$
2	$a1$	$a1.c1$	$a1.c1.b2$
3	$a1$	$a1.c1$	$a1.f2.c1.b1$
4	$a1$	$a1.c1$	$a1.f2.b2$
5	$a1$	$a1.f2.c1$	$a1.c1.b1$
6	$a1$	$a1.f2.c1$	$a1.c1.b2$
7	$a1$	$a1.f2.c1$	$a1.f2.c1.b1$
8	$a1$	$a1.f2.c1$	$a1.f2.b2$

Fig. 12. Sample XML document for Example 4.5, and view content on this document.

t_i is identical to t except that t_i contains a symbol of the form R_a where t contains Δ_a^-, for some a. Observe that each of the terms $t_1, t_2, \ldots, t_k$ have $k-1$ Δ^- relations, and $k-1$ is odd, therefore, by claim (i), they all represent tuples to remove from v. Thus, each tuple which t would attempt to add as a result of the deletion u, is deleted at least k times with $k \geq 2$, whereas t would attempt to add it only once. In other words one does not actually need to develop the positive ($\cup$) terms in the deletion expression at all, since that data would be removed "more times than t can add it". Thus, we can simply ignore such positive terms. □

Example 4.4. Consider a view v_2 defined as $//a_{ID}[//c_{ID}]//b_{ID}$. Elementary development of the deletion expression shows that the delete expression of v (prior to any term pruning) is:

$$R_aR_bR_c \setminus \Delta_a^- R_bR_c \setminus R_a\Delta_b^- R_c \setminus R_aR_b\Delta_c^- \cup \underline{\Delta_a^-\Delta_b^- R_c} \cup \underline{\Delta_a^- R_b\Delta_c^-} \cup \underline{R_a\Delta_b^-\Delta_c^-} \setminus \Delta_a^-\Delta_b^-\Delta_c^-$$

where the underlined terms are those prefixed with $\cup$, whose tuples should be "added" to reflect a deletion. Let us consider these terms one by one:

— $\Delta_a^-\Delta_b^- R_c$ tuples are deleted first, by the term $\Delta_a^- R_bR_c$, and second, by the term $R_a\Delta_b^- R_c$;
— $\Delta_a^- R_b\Delta_c^-$ tuples are deleted first, by the term $\Delta_a^- R_bR_c$, and second, by the term $R_aR_b\Delta_c^-$;
— $R_a\Delta_b^-\Delta_c^-$ tuples are deleted first, by the term $R_aR_b\Delta_c^-$ and second, by the term $R_a\Delta_b^- R_c$.

Proposition 4.3 allows reducing this expression to:

$$R_aR_bR_c \setminus \Delta_a^-\Delta_b^-\Delta_c^- \setminus \Delta_a^- R_bR_c \setminus R_a\Delta_b^- R_c \setminus R_aR_b\Delta_c^-$$

In this expression, data is only *removed* from $R_aR_bR_c$, which rejoins the intuition that deletes should not add tuples to our (conjunctive, monotone) views.

The following example illustrates the cummulated impact of the Propositions 4.2 and 4.3.

Example 4.5. Consider the view v_2: $//a_{ID}[//c_{ID}]//b_{ID}$ and the XML document d shown on the left in Figure 12. The tuples of the view v_2 evaluated on d appear in the same Figure on the right. Consider the update u_2 deleting $//a/f/c$ from d. This amounts to deleting the d subtree rooted in the node identified by $a1.f2.c1$. The full deletion expression of v_2 under the update u_2 (as in Example 4.4) is:

$$R_aR_bR_c \setminus \Delta_a^- R_bR_c \setminus R_aR_b\Delta_c^- \setminus R_a\Delta_b^- R_c \cup R_a\Delta_b^-\Delta_c^- \cup \Delta_a^- R_b\Delta_c^- \cup \Delta_a^-\Delta_b^- R_c \setminus \Delta_a^-\Delta_c^-\Delta_b^-$$

Proposition 4.2 eliminates the second, sixth and seventh join terms from this expression, since they are guaranteed to be empty. Thus, the deletion expression is reduced to:

$$R_aR_bR_c \setminus R_aR_b\Delta_c^- \setminus R_a\Delta_b^- R_c \cup R_a\Delta_b^-\Delta_c^- \setminus \Delta_a^-\Delta_c^-\Delta_b^-$$

Proposition 4.3 eliminates the positive term, and the deletion expression becomes:

$$R_aR_bR_c \setminus R_aR_b\Delta_c^- \setminus R_a\Delta_b^- R_c \setminus \Delta_a^-\Delta_c^-\Delta_b^-$$

The computation of the pending update list leads to discovering that $\Delta_a^- = \emptyset$, which further simplifies the deletion expression to:

$$R_a R_b R_c \setminus R_a R_b \Delta_c^- \setminus R_a \Delta_b^- R_c$$

The pending update list also informs us that $\Delta_b^- = \{(a1.f2.c1.b1)\}$ and $\Delta_c^- = \{(a1.f2.c1)\}$. Thus, $R_a R_b \Delta_c^-$ contains the tuples $5, 6, 7$ and 8 from the view, while $R_a \Delta_b^- R_c$ consists of the tuples 3 and 7 from the view. The update u thus reduces v to its tuples $1, 2$ and 4.

4.3. Term pruning based on Delta tables

In this Section we discuss term pruning from the delete expression, based on the Δ^- tables computed together with the pending update list. Consider the following example:

Example 4.6. Let v be the view $//c_{ID}//b_{ID}$ and d be the XML document shown in Figure 11. Consider an update u_3 deleting $//f$, which in d targets the node identified by $a1.f2$. As a side effect of the deletion, the node identified by $a1.f2.b1$ is also removed, thus $\Delta_b^- = \{(a1.f2.b1)\}$. From the identifier of the single Δ_b^- node, we see that this node does not have a c ancestor. Therefore, the deletion expression term $R_c \Delta_b^-$ is empty.

This example generalizes into:

PROPOSITION 4.7. *Let v be a view and n_1, n_2 be two nodes in v, such that n_2 is a $/$ or $//$ child of n_1 in v. Consider an update u and let $\Delta_{n_2}^-$ be the delta table corresponding to the label of n_2. If for every node $m \in \Delta_{n_2}^-$, the ID of node m shows that m has no ancestor labeled n_1, then all terms in the v deletion expression containing $R_{n_1} \Delta_{n_2}^-$ are empty.*

4.4. Delete propagation algorithms

The general algorithm Propagate Delete by Deleting Tuples (or **PDDT**, in short), computing the view tuple deletions that result from XML tree deletions, is outlined in Algorithm 5.

ALGORITHM 5: Propagate Delete by Deleting Tuples **(PDDT)**

Input: view v, delete update u
Output: updated view v' to reflect u
Compute the Δ^- tables corresponding to u
Expand the deletion expression to 2^k terms and prune them independently of u (Propositions 4.2 and 4.3) and based on the Δ^- tables (Proposition 4.7)
$\Delta_v^- \leftarrow \emptyset$ (tuples to be possibly deleted from v)
foreach *term t surviving pruning* **do**
  evaluate t (use materialized snowcaps, Δ^- tables, structural joins etc.) and add its results to Δ_v^-
foreach *tuple $t_v \in$ view v* **do**
  if $t_v \in \Delta_v^-$ **then**
    remove t_v from v
  else
    if *t_v stores ID, val or $cont$ for a node n (i) appearing in some Δ^- table or (ii) having an ancestor in some Δ^- table* **then**
      decrease t_v derivation count
      if *t_v derivation count becomes* 0 **then**
        remove t_v from v
If needed, update auxiliary structures

Algorithm **PDDT** covers both tuple deletions, and decreasing the derivation count of a tuple while not necessarily removing it from the view. Both cases are illustrated by the following example:

Example 4.8. To illustrate the first case, consider the view $//a_{ID}//b$, the document d in Figure 11, and an update deleting $//c//b$. The view contains two tuples corresponding to node b. The

deletion removes the b node identified by $a1.c1.b1$, which belongs to Δ_v^-. This case turns to be the same as the one illustrated by Example 4.1. Therefore, no check of the derivation count is necessary as t_v matches exactly the node in Δ_v^-.

To illustrate the second case, consider the view $//a_{ID}[//b]$, the document d in Figure 11, and an update deleting $//c//b$.

The view contains a single tuple corresponding to node a. The tuple has a derivation count of 2 due to the two b nodes matching the existential view branch. The deletion removes the b node identified by $a1.c1.b1$, but this still leaves a b descendant to the a node. Therefore, the update decreases the derivation count of the view tuple $(a1)$ by one unit, setting it to 1.

Now consider a second update, deleting $//f//b$. This will lead to removing the node $(a1.f2.b1)$, reducing the derivation count of the corresponding v tuple to 0 and thus removing the tuple from the view.

A class of deletion impacts not considered so far is when the update *modifies* view tuples, by altering some val or $cont$ attribute that the view stores. (According to the W3C XML data model [XQuery Data Model 2010] adopted in this study, node identity is immutable, thus node IDs are never modified). Such modifications occur when an update deletes tuples or modifies derivation counts.

We have designed an algorithm **PDMT** (Propagate Delete by Modifying Tuples) which can be seen as symmetric to Algorithm 4 (PIMT, focusing on tuple modifications due to insertions). Considering that modifications, deletions and derivation count updates may all occur due to the same deletion, for generality, we provide a general algorithm (called **(PDDT/MT)**) handling all these possibilities. Algorithm **PDDT/MT** is the one which we actually run, and it incorporates the main steps of PDDT and PDMT (computing the pending update list only once, then performing the processing of PDDT and PDMT, and finally updating the auxiliary data structures only once). Algorithm **PDDT/MT** is outlined in Algorithm 6. The main steps of the delete propagation procedure are summarized in Figure 9 (with Figure 10 explaining the acronyms).

PROPOSITION 4.9 (COMPLEXITY OF ALGORITHMS 5 AND 6). *Let v be a view of k nodes, $|v|$ denote the number of tuples in v, and u be a deletion resulting in the pending update list PUL. Let R be the largest relation among the leaf nodes in v's lattice. The complexity of Algorithm 5 (PDDT) is $\mathcal{O}(2^k \times k \times max(|PUL|, |R|) + |v| \times s(\Delta^-))$, where $s(\Delta^-)$ is the cost to search for a tuple in one of the Δ^- tables, containing the ID of a node appearing in the view, or of an ancestor of such a node (check performed in Algorithm 5). As for PINT and PIMT, $2^k \times k \times max(|PUL|, |R|)$ is a bound for the cost of evaluating all the deletion terms which have survived pruning. The second term $|v| \times s(\Delta^-)$ corresponds to the PDDT loop over v tuples. The complexity of Algorithm 6 (PDDT/MT) is $\mathcal{O}(2^k \times k \times max(|PUL|, |R|) + |v|^2 \times s(\Delta^-))$.*

5. OPTIMIZING THE PROPAGATION OF XML UPDATE SEQUENCES

In the previous Sections we have provided algorithms for propagating an update to a materialized view. These algorithms can be applied after each individual update statement (immediate propagation). However, when a sequence of updates is applied to the document, their propagation to the views may be deferred, and possibly applied in lazy mode, i.e., only when the view data is consulted by a query.

Recent work [Cavalieri et al. 2011] has focused on the efficient handling of multiple updates and multiple Pending Update Lists (PULs), and provided a set of rules for making their evaluation more efficient. We recall that a PUL is a sequence of atomic update operations that are applied to source documents. In our work, we rely on PULs to handle the propagation of source modifications to views. Therefore, it seemed quite natural to apply the optimization techniques presented in [Cavalieri et al. 2011] to view maintenance. Notably, they have provided *reduction* rules to remove useless update operations; *conflict* rules to determine if there exists a conflict between PULs to be run in parallel, along with a framework for specifying conflict resolution policies when there are conflicts;

ALGORITHM 6: Propagate Delete by Deleting/Modifying Tuples **(PDDT/MT)**

Input: view v, delete update u
Output: updated view v' to reflect u
Compute the Δ^- tables corresponding to u
Develop the 2^k set difference terms to be deleted from v in case of deletions and prune terms based on the Δ^- tables (Propositions 4.2, 4.3 and 4.7)
$\Delta_v^- \leftarrow \emptyset$ (tuples to be possibly deleted from v)
$cvn \leftarrow \{n \in v, n$ annotated with $cont$ or $val\}$;
foreach *term t surviving pruning* **do**
  evaluate t (use materialized snowcaps, Δ^- tables, structural joins etc.) and add its results to Δ_v^-
if $\Delta_v^- \neq \emptyset$ **then**
  foreach *tuple $t_v \in$ view v* **do**
    if $t_v \in \Delta_v^-$ **then**
      decrease t_v derivation count
    if t_v *derivation count becomes 0* **then**
      remove t_v from v
      if $t_v.n \in cvn$ **then**
        remove $t_v.n$ from cvn
    else
      if $cvn \neq \emptyset$ **then**
        call Algorithm **PDMT**(v,u)
else
  if $cvn \neq \emptyset$ **then**
    call Algorithm **PDMT**(v,u)
If needed, update auxiliary structures

and *aggregation* rules for combining PULs to be run sequentially. In particular, their techniques do not need to access the source document, enabling such optimized handling of PULs to take place remotely from where the source documents are stored. As such, they could be easily integrated with our view maintenance algorithms, as the latter would use the optimized PULs instead of the original ones.

In this Section, we show how to apply the optimization rules of [Cavalieri et al. 2011] for the different purpose of efficiently propagating source updates to the materialized views.

5.1. Sequencing updates in view maintenance

Figure 13 shows the interaction between the optimization framework in [Cavalieri et al. 2011] and our framework. The rules in [Cavalieri et al. 2011] cannot be directly applied to our statement-level updates, as they are defined on atomic operations. For such a reason, we needed to interleave our algorithms and theirs in the following way. We start with a sequence of statement-level updates, which is reduced by our view maintenance algorithms into a sequence of atomic node-level updates. Such atomic updates are then submitted to the optimization process as described in their paper (and detailed below). The optimized sequence of atomic updates is then propagated to the view (after being evaluated on the original document as well), instead of the original sequence of atomic updates. Figure 13 shows that we call the Algorithm PINT/PIMT or PDDT/PDMT, with optimized sequences of atomic updates to be applied to the view. Precisely, for each insert/delete in a sequence of statement-level updates, we produce atomic updates by **CP** (compute-pul), we reduce them with the optimization rules **OR**, and run them in the order of appearance.

In the experimental study, we have reimplemented their optimization rules and designed a set of experiments showing the gain of applying the reduced sequence of updates to the view versus applying the original sequence to it.

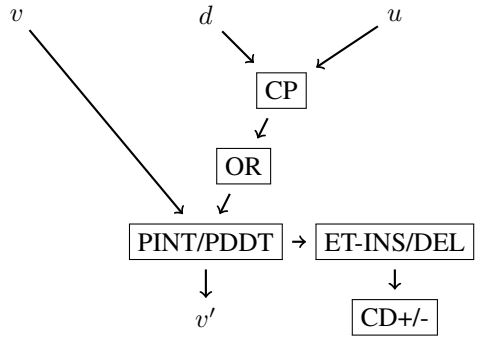


Fig. 13. Interleaving PINT/PDDT with Optimization Rules (OR).

O1)	$\frac{op_1{=}op(n,\text{-})op_2{=}op'(n,\text{-})}{op_1,op_2 \nabla_1 op_2}$ $\begin{matrix} op \in \{del, ins\searrow\} \\ op' \in \{del\} \end{matrix}$	If there is an insertion or deletion, followed by a deletion on the same target node, then just perform the second deletion.
O3)	$\frac{op_1{=}op(n,\text{-})op_2{=}op'(n',\text{-})}{op_1,op_2 \nabla_1 op_2}$ $\begin{matrix} op \in \{ins\searrow, del\} \\ op' \in \{del\} \end{matrix}$ $n_{//_d}n'$	If there is an insertion or a deletion, followed by a deletion on an ancestor of the first operation, then just perform the second operation.
I5)	$\frac{op_1{=}op(n,L_1)op_2{=}op(n,L_2)}{op_1,op2 \nabla_1 op(n,[L_1,L_2])}$ $c(op) = i$	If there is an insertion, followed by another insertion on the same target node, then combine the insertions into one.

Fig. 14. Reduction Rules.

5.2. Update Operations

Eleven elementary update operations are handled in [Cavalieri et al. 2011]. For brevity, we will consider here two fundamental update operations:

— insert a forest P after the last child of a node v, denoted $ins\searrow(v, P)$;
— delete a node v, denoted $del(v)$.

5.3. Rules

The rules from [Cavalieri et al. 2011] applied to the two operations we consider are detailed below. Recall that reduction rules allow simplifying the update sequence. If the update sequences are to be executed in parallel then the lists can be integrated, however, as conflicts can occur there are a set of conflict rules. Finally, if sequences are to be executed sequentially then the aggregation rules can be used.

5.3.1. Reduction Rules. Twenty reduction rules are specified in [Cavalieri et al. 2011], divided into nine stages. Rules within the same stage can be evaluated in any order, however, the rules of stage n can only be evaluated after the rules of all previous stages $1, 2, \ldots, n-1$. The rules applying to our operation set are detailed in Figure 14.

In the above operations, the ∇ operator reduces a sequence of input operations. ∇_1 indicates the membership of such rules to the first stage. In rule O3, $n_{//_d}n'$ defines that n is a descendant of n'. Within operation I5, $c(op) = i$ specifies the applicability of the rule to the various classes of insertions handled in their paper. In our specific case, $c(op) = ins\searrow$.

5.3.2. Conflict Rules. For update sequences to be executed in parallel a method for integrating them has been created. However, conflicts can arise. Therefore, conflict rules have been specified. The rules can pick up the following conflicts: repeated modifications; repeated attribute insertion;

IO) $\frac{t(op_1)=t(op_2)\ o(op_1)=o(op_2)\ \epsilon\ \{ins^{\searrow}\}}{op_1 \overset{3}{\leftrightarrow} op_2}$

This identifies the case where operations are different depending on execution order.

LO) $\frac{t(op_1)=t(op_2)\ o(op_1)\ \epsilon\ \{del\} o(op_2)\ \epsilon\ \{ins^{\searrow}\}}{op_1 \overset{4}{\rightarrow} op_2}$

This identifies the case when an operation is overridden by another with the same target node. Which causes the first operation to not affect the document due to the presence of the second one.

NLO) $\frac{t(op_2)//_d t(op_1)\ o(op_1)\ \epsilon\ \{del\}\ o(op_2)\ \epsilon\ \{ins^{\searrow}\}}{op_1 \overset{5}{\rightarrow} op_2}$

This identifies the case when an operation is overridden by another with a different target node. This case arises when the deletion's target node is an ancestor of the insertion's target node.

Fig. 15. Conflict Rules.

A1) $\frac{op_1=op(v,L_1)\ op_2=op(v,L_2)}{\Delta_1'\cup\{op_1,op_2\}, \Delta_2' \overset{A}{-\circ_1} \Delta_1'\cup\{op(v,[L_1,L_2])\}, \Delta_2'}\ c(op)=i$

If both the insertions are on the same node then aggregate the PULs by adding the second insert to the first PUL, after the first insert.

A2) $\frac{op_1=op(v,L_1)\ op_2=op(v,L_2)}{\Delta_1', \Delta_2'\cup\{op_1,op_2\} \overset{A}{-\circ_1} \Delta_1', \Delta_2'\cup\{op(v,[L_1,L_2])\}}\ c(op)=i$

This is A1 in reverse. Add the insert from the first PUL into the second PUL before the second insert. .

D6) $\frac{op_1=op(v,T_1\cdots T_i\cdots T_n)\ \Delta_{v^i}=\{o\ \epsilon\ \Delta_2 \mid t(o)=v'\}}{\Delta_i'\cup\{op_1\}, \Delta_2 \overset{A}{-\circ_4} \Delta_1'\cup\{op(v,P)\}, \Delta_2\backslash\Delta_{v'}}\ \underset{\substack{T_i \models \Delta_{v'}\ leads\ to\ T_i' \\ P=T_1\cdots T_i'\cdots T_n}}{v'\ \epsilon\ V(T_i),}$

This identifies the case where operations in the second PUL reference node(s) of a tree which is a parameter of an operation in the first PUL. In this case the operations in the second PUL, referencing the tree, are performed and removed from the second PUL.

Fig. 16. Aggregation Rules.

element insertion order; local override; and non-local override. The conflict rules that are applicable to our work, based on our set of supported operations, are detailed in Figure 15.

The first rule IO (Insertion Order) states the interchangeability of two operations of the same kind with the same target. This conflict is symmetric as indicated by the substitutability symbol $\overset{3}{\leftrightarrow}$, where 3 is the rule order in their paper, which we have maintained for compatibility. Conversely, the rules LO (Local Override) and (Non Local Override) are asymmetric, as the first is the overriden operation and the second the overriding one, thus motivating the asymmetric symbol.

After applying the conflict rules to the update sequence a list of non-conflicting operations are returned along with the identified conflicts. The work allows PUL producers to define conflict resolution policies in order to solve conflicts during PUL integration.

5.3.3. Aggregation Rules. For update sequences to be executed sequentially a method for aggregating them has been created. Aggregation involves merging multiple sequences. Precisely, given two sequences Δ_1 and Δ_2, their aggregation into a single sequence Δ corresponds to the sequential execution of $\Delta_1;\Delta_2$, where Δ_1 is applied to the original document and Δ_2 is applied to the document updated through Δ_1. A set of aggregation rules can be applied to aggregate operations throughout the two sequences. Again, not all these rules apply to our work, the ones that do are detailed in Figure 16.

In the above rules, $\overset{A}{-\circ}$ denotes the aggregation operator applied to two PULs, and the associated number specifies which stage the operations belong to.

5.4. Examples

The following examples show how the reduction, integration and aggregation of PULs are handled. We represent the PULs in our syntax, i.e., by making the IDs of nodes explicit, exactly as our view

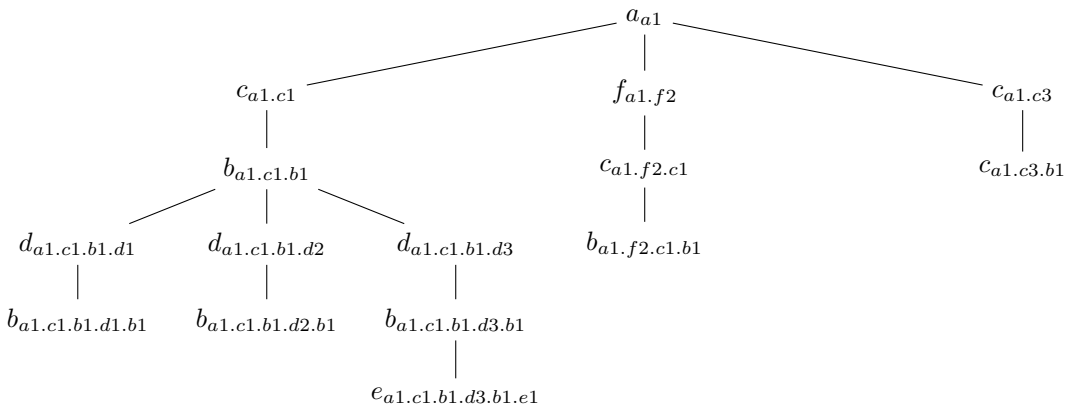


Fig. 17. Sample XML document.

maintenance framework encodes them. Example 5.1 shows how a PUL can be reduced. Example 5.2 shows how two sequential PULs can be integrated. Finally, Example 5.3 shows how two parallel PULs can be aggregated.

Example 5.1. Let Δ be the PUL specified on the document doc in Figure 17 containing the following operations:
$op_1 = ins \searrow (a1.c1.b1.d1.b1, \langle b\rangle\langle d/\rangle\langle/b\rangle)$,
$op_2 = del(a1.c1.b1.d1.b1)$,
$op_3 = ins \searrow (a1.c1.b1.d2.b1, \langle b/\rangle)$,
$op_4 = del(a1.c1.b1.d2)$,
$op_5 = ins \searrow (a1.c1.b1.d3, \langle b/\rangle)$,
$op_6 = ins \searrow (a1.c1.b1.d3, \langle d\rangle\langle b/\rangle\langle/d\rangle)$

Let v be the view $//b//d//b$ over doc. The reduced PUL is $\{del(a1.c1.b1.d1.b1), del(a1.c1.b1.d2), ins \searrow (a1.c1.b1.d3, \langle b/\rangle, \langle d\rangle\langle b/\rangle\langle/d\rangle)\}$. This is because op_1 is ignored due to rule O1; op_3 is ignored due to rule O3; and op_5 and op_6 are combined due to rule I5.

Example 5.2. Let $\Delta_1 = \{op_1^1 = ins \searrow (a1.c1.b1.d1, \langle d\rangle\langle b/\rangle\langle/d\rangle), op_1^2 = del(a1.c1.b1.d2), op_1^3 = del(a1.c1.b1.d3)\}$, $\Delta_2 = \{op_2^1 = ins \searrow (a1.c1.b1.d1, \langle b/\rangle), op_2^2 = ins \searrow (a1.c1.b1.d2, \langle b/\rangle), op_2^3 = ins \searrow (a1.c1.b1.d3.b1, \langle b/\rangle)\}$ be the PULs over document doc in Figure 17. Let v be the view $//b//d//b$ over doc.

Aggregation does not apply in this example, as every operation causes a conflict. op_1^1 and op_2^1 are in conflict due to the insertion order (IO) rule; op_1^2 and op_2^2 are in conflict due to the local overriding (LO) rule; and op_1^3 and op_2^3 are in conflict due to the non-local overriding (NLO) rule. How these conflicts are solved depend on the conflict resolution policies the PUL producers specify. The algorithm will fail if it cannot identify a valid reconciliation, that is: a PUL with no conflicts, satisfying the policies of all the PUL producers involved.

Example 5.3. Let $\Delta_1 = \{op_1^1 = ins \searrow (a1.c1.b1.d1.b1, \langle c\rangle\langle b/\rangle\langle/c\rangle), op_1^2 = ins \searrow (a1.c1.b1.d2, \langle b/\rangle), op_1^3 = ins \searrow (a1.c1.b1.d3, \langle d\rangle\langle b/\rangle\langle/d\rangle)\}$, $\Delta_2 = \{op_2^1 = ins \searrow (a1.c1.b1.d1.b1, \langle b/\rangle), op_2^2 = ins \searrow (a1.c1.b1.d2, \langle d\rangle\langle b/\rangle\langle/d\rangle), op_2^3 = ins \searrow (a1.c1.b1.d3.b1, \langle b/\rangle)\}$. Let v be the view $//b//d//b$ over doc.

Aggregation $\Delta_1 \mapsto \Delta_2$ is $\{op_1^1 = ins \searrow (a1.c1.b1.d1.b1, \langle c\rangle\langle b/\rangle\langle /c\rangle, \langle b/\rangle), op_2^2 = (a1.c1.b1.d2, \langle b/\rangle, \langle d\rangle\langle b/\rangle\langle /d\rangle), op_1^3 = ins \searrow (a1.c1.b1.d3, \langle d\rangle\langle b/\rangle\langle b/\rangle\langle /d\rangle)\}$. This is because op_1^1 and op_2^1 have the same target node, so the XML insertion fragments are combined into the one operation due to rule A1; op_1^2 and op_2^2 are combined due to rule A2 (A1 in reverse); and op_1^3 and op_2^3 are combined due to rule D6. The XML fragment for the final insertion operations are combined due to the second fragment being an insertion on a node of the first XML fragment to be inserted.

6. EXPERIMENTAL EVALUATION

In this Section, we present a set of experiments that gauge the effectiveness of our techniques for XML incremental view maintenance.

Section 6.1 describes the experimental setting. Section 6.2 studies the performance of our algorithms, showing how the running times decompose across the different view maintenance steps. Section 6.3 assesses the impact on performance of the syntactic complexity of views and updates, whereas Section 6.4 is concerned with scalability when document size varies. Section 6.5 compares our approach with fully recomputing the view and Section 6.6 compares it with the closest competitor [Sawires et al. 2005]. Section 6.7 studies the impact of the choice of snowcaps used for maintenance, through two simple sample choices. Finally, Section 6.8 compares performance with and without the use of a set of dynamic reasoning pruning rules.

6.1. Settings

We have implemented the PINT, PDMT and PDDT/MT algorithms described in this paper using Java 6, within the ViP2P Java-based platform (http://vip2p.saclay.inria.fr) developed at Inria. ViP2P provides our implementation of tree patterns from the $\mathcal{P}$ dialect used in this work, the Compact Dynamic Dewey ID implementation, as well as an execution engine providing the usual operators (selections, projections, hash-joins etc.) and XML-specific structural joins. ViP2P stores view data within BerkeleyDB v4.0.71. For some operations (see below), we rely on the widely known Saxon XQuery processor v9.2.1.1j. Unless stated otherwise, our experiments ran on a MacBook Pro with Mac OS X 10.6.7, a 2.8 GHz Intel Core 2 duo processor and 4GB memory. Experiments in Sections 6.3, 6.7 and 6.8 were run on a PC with Linux Kubunto v2.6, with a Pentium 4 260GHz CPU and 1GB memory.

Documents, views and queries We use XMark [Schmidt et al. 2002] benchmark documents of different sizes. As in [Benedikt and Cheney 2010], we use queries from the (read-only) XMark benchmark as views, and a set of updates derived from the XPathMark benchmark [Franceschet 2005] by inserting dummy elements into each of (or deleting, respectively) the nodes returned by the respective XPathMark query. While we aimed to use the same queries and updates as in [Benedikt and Cheney 2010], their focus was on testing whether the view and the update were *independent*, which was the case in most of their examples. In contrast, we are interested in the cases updates do affect views. Therefore, we added extra updates, characterized by path expressions from the A, B and E subsets of the XPathMark benchmarks; the names of these queries start with the respective letter. When these views and updates used features of the language not covered by ours (Section 2), we used simplified versions which did fit our language. Finally, to complete the illustration, we also added a set of path updates of our invention, whose names start with X.

We report the results obtained with XML insertions, i.e., running algorithms **PINT** (Section 3.5) and **PIMT** (Section 3.6) and the results obtained with XML deletions, running algorithms **PDDT/PDMT** (Section 4).

Implementation details In the algorithms, we use Saxon for the first step of our approach, namely identifying the target insert/delete nodes, that will receive respectively new children or be removed. To actually update the document, we build the pending update list, add the new children or remove the target node and its descendants using Saxon's in-memory operations, and then serialize the modified document again using Saxon. To update the view, we extract the Δ^+ or Δ^- tables,

respectively for PINT and PDDT algorithms, from the pending update list and apply all the respective steps described in Section 3 and Section 4. All algebraic operations (notably joins) are performed using ViP2P's physical operator library. We stress that ViP2P is a Java-based prototype and improvements by constant factors could probably be obtained by further optimizing the code, using C++ etc. Nevertheless, we implemented all algorithms in the same framework, and relied on state-of-the art algorithms, e.g., for structural joins etc. Thus, we believe our experiments accurately reflect the performance trade-offs involved.

Measured times In the following, we report on a set of times which were averaged over five executions. *Find Target Nodes* is the time taken by Saxon to identify the nodes affected by an update/deletion operation. *Compute Delta Tables* is the time taken to build the Δ^+ or Δ^- tables starting from the target nodes and the inserted/deleted XML fragments. *Get Update Expression* is the time to build, unfold, and prune the algebraic expression corresponding to the update(s)/deletion(s) to be propagated. *Execute Update* is the time to evaluate the expression obtained within ViP2P's algebraic XML query evalution engine using the lattice and the time to add/remove the resulting tuples to the database. *Update Lattice* is the time to update the auxiliary data structures (lattice terms).

6.2. Performance of the incremental maintenance algorithms

Our first experiments used as views the queries Q_1, Q_2, Q_3, Q_4, Q_6, Q_{13} and Q_{17} of the XMark [Schmidt et al. 2002] benchmark. For each query, we have employed a set of update path expressions (as described above), divided into five classes, each sharing a characteristic that provides a letter (underlined next) for us to identify them: (c1) Linear path expressions; (c2) path expressions with an And predicate; (c3) path expressions with an Or predicate; (c4) path expressions with an AO (and-or) predicate; (c5) Linear Boolean path expressions.

Figures 18 and 19 show for each pair (view, update) and (view, deletion), respectively, the view maintenance time, broken down into its individual components. It can be observed that in all cases the times to *Compute Delta Tables*, *Get Update Expression* and *Execute Update* are smaller than the time to locate the target nodes, which, as expected, depends on the corresponding target XPath expression. Moreover, the *Update Lattice* is more impacted by the complexity of the view considered and the document size, than by the specific update applied. For views like Q_3 (FLWR expression with conditions), it almost stays steady while increasing the complexity of the update path expression from class (c1) to (c5).

In Figure 19, the time to *Get Update Expression* is smaller than the corresponding time in Figure 18 for all three views since the pruning is much faster in the case of deletions. The time to *Execute Update* in Figure 19 is affected in particular by:

— the total number of target nodes and their descendants: the higher this number, the higher the update propagation time;
— the update of val and $cont$ attributes of the nodes affected by the deletion, since such nodes trigger PDMT execution which in turn is expensive;
— the total number of the terms surviving the pruning in the deletion expression, since their computation is also quite expensive.

As an illustration of the second point above, consider an example of the execution of the PDMT algorithm, with a view like Q_3 and deletions like $X3\_A$ and $X4\_O$, in which the time to *Execute Update* depends on the number of view tuples to be removed detected by the PDDT Algorithm. The execution of the deletion expression identifies a total of 2889 tuples ($t_\Delta \in \Delta_v^-$) to be removed out of a total of 6182 tuples of the initial view. Among the 2889 tuples, 15 are directly removed (by the PDDT algorithm). Then, the PDMT algorithm needs to be invoked on the remaining 6127 ones; it verifies whether, for each view tuple, and for all nodes with val or $cont$ attributes, the deletion impacts the respective attributes or not. As shown by this example, the fewer the tuples to be removed by the PDDT algorithm, the more expensive the execution of the PDMT algorithm.

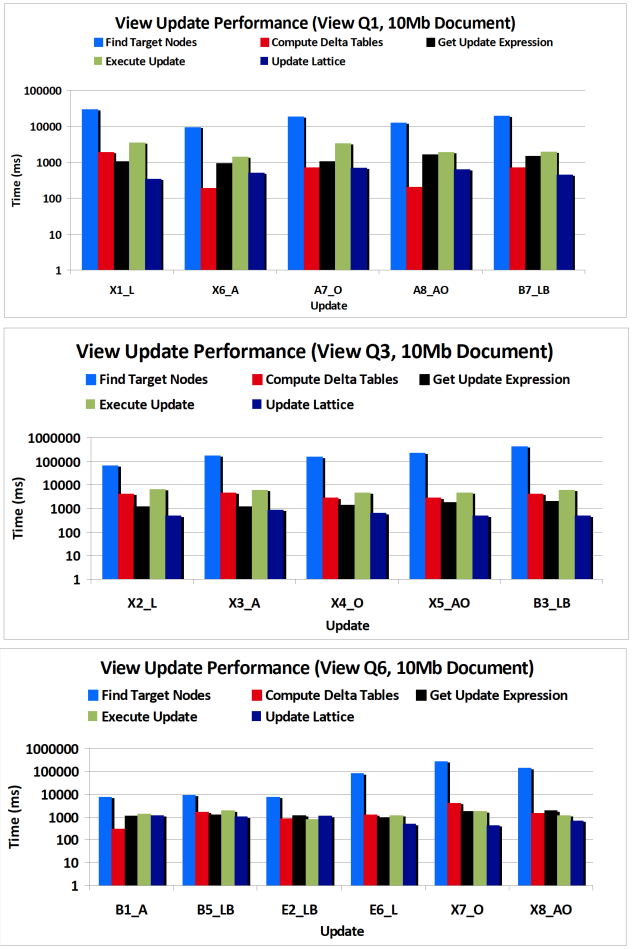


Fig. 18. PINT and PIMT algorithms: time breakdown for insert propagation to XMark views Q_1 (top), Q_3 (middle), and Q_6 (bottom).

The impact of the last parameter (the number of terms surviving the pruning of the deletion expression) can be understood by recalling that the term computation is an important component of the PDDT cost and complexity (Proposition 4.9).

Comparing the time breakdowns for insertions and deletions in Figure 18 and Figure 19, we notice that the time to *Update Lattice* for the PDDT is greater than the time to *Update Lattice* for the PINT algorithm. This is due to the fact that inserting new tuples just requires inserting the tuples into the lattice, whereas deleting tuples requires searching the lattice for the tuples to be removed.

Concluding, Figures 20 and 21 report the performance results for all the XMark views considered, by summing up the *Find Target Nodes*, *Compute Delta Tables*, *Get Update Expression*, *Execute Update* and *Update Lattice* times for all queries.

6.3. Impact of the syntactic complexity of views and updates

We have run experiments to assess the impact of the syntactic complexity of updates. More specifically, we consider an update $X1\_L$ deleting all nodes on an XPath path p, thus, algorithms PDDT/PDMT are involved. We vary the complexity of p (the number of steps on the path). The

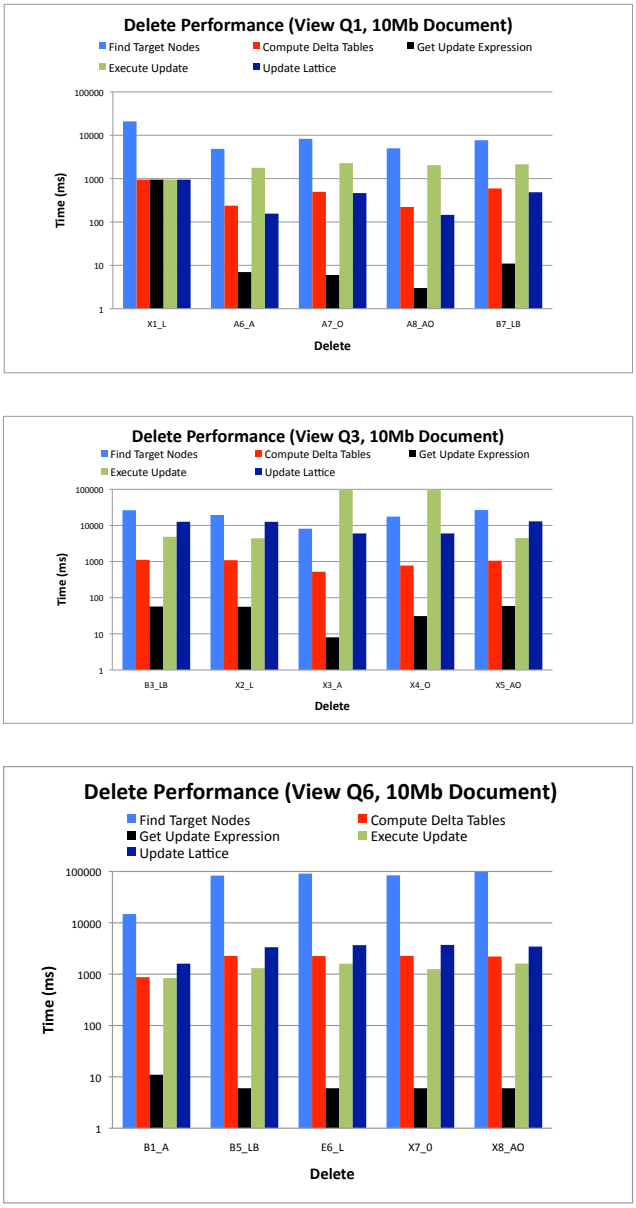


Fig. 19. PDDT/MT algorithms: time breakdown for delete propagation to XMark views Q_1 (top), Q_3 (middle), and Q_6 (bottom).

experiment uses the view $Q1$, and a database of 100KB and 10MB; the results are shown in Figures 22 and 23.

One can see that the view maintenance time decreases as the update path lengthens, which is to be expected since shorter paths to the deletion nodes mean more nodes are being deleted. In our approach, this translates into an increasing number of non-empty Δ^- tables as the update path length decreases.

We performed a separate experiment to assess the impact of the view node annotations (that is, of the data items stored in each view node) on the performance of the incre-

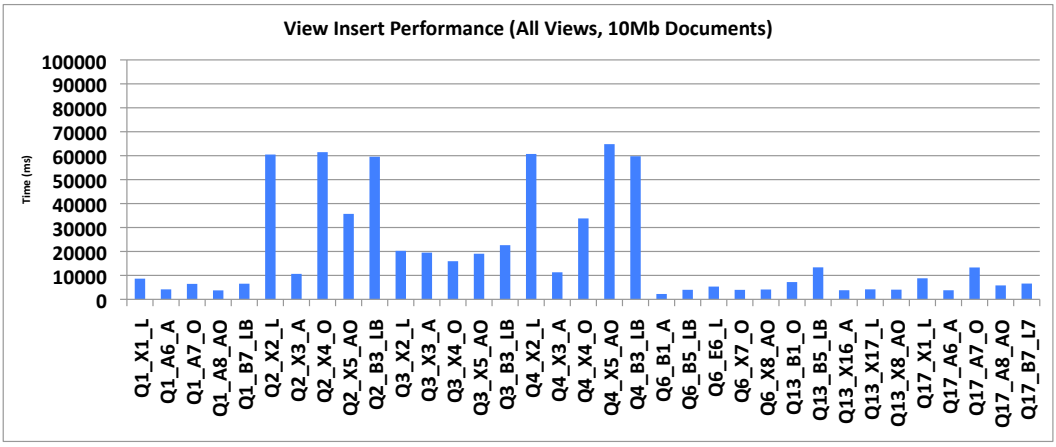


Fig. 20. Running time of the PINT algorithm for all the XMark views.

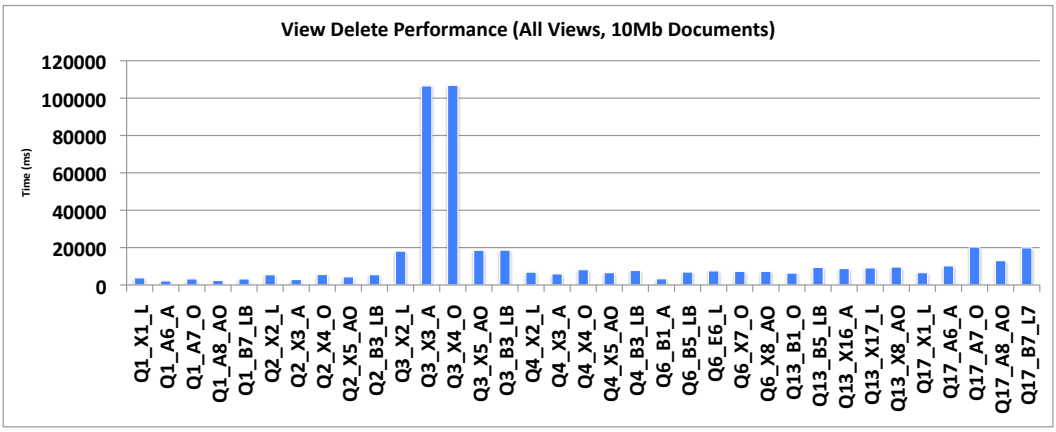


Fig. 21. Running time of the PDDT algorithm for all the XMark views.

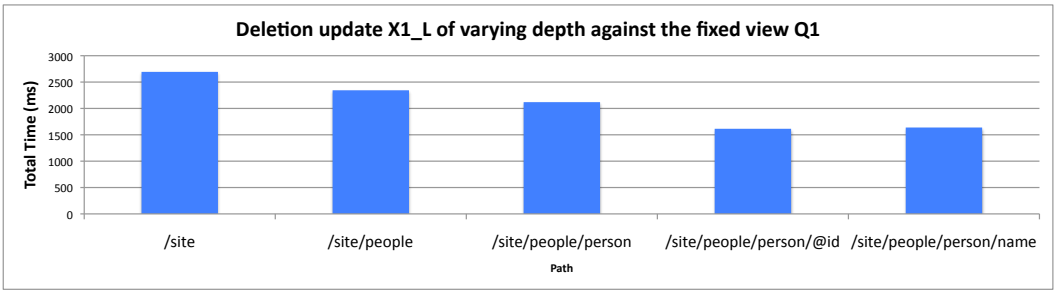


Fig. 22. Varying Paths for XMark view Q_1 and update $X1\_L$ on a 100KB database.

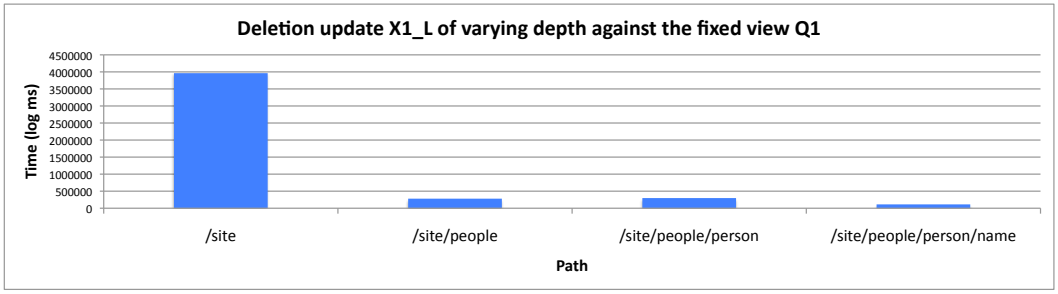


Fig. 23. Varying Paths for XMark view Q_1 and update $X1\_L$ on a 10MB database.

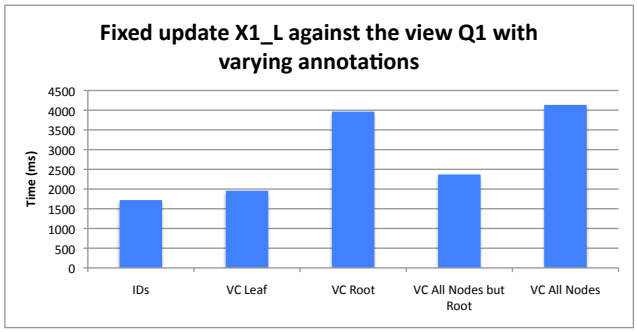


Fig. 24. Varying Annotations for XMark view Q_1 and update $X1\_L$.

mental maintenance algorithm. In this setting, we applied a fixed update $X1\_L$ removing /site/people/person[@id="person0"] on a set of variants of the same view $Q1$ (corresponding to /site$_{ID}$/people$_{ID}$/person$_{ID}$[@id$_{ID}$]/name$_{ID,Val}$. The selection predicate is included in $X1\_L$ so the update will result in deletions and modifications. In particular, with such a variant of $X1\_L$ the recursive calls of PDMT in Algorithm 6 will be fired on the view ancestors of the person nodes.

The variants are chosen so that they would significantly differ in their annotations. All the variants store IDs for all nodes. In the first view variant, denoted IDs, nothing else is stored. The second variant is denoted VC_Leaf and stores val and $cont$ annotations for the name leaf node. In VC_Root, val and $cont$ are stored only for the root, while in $VC\_All\_Nodes\_but\_Root$, val and $cont$ are stored for every node but the root. Finally, in VC_All_Nodes, val and $cont$ are stored in every view node.

Figure 24 depicts the results. We notice that the closer val and $cont$ are to the root of the view, the greater the evaluation time for PDDT/PDMT as PDMT has to work with larger val and $cont$. When val and $cont$ are pushed towards the leaves, the evaluation time is smaller, justified by the same intuition (smaller values of val and $cont$ in the view tuples, into which one needs to search and apply updates).

6.4. Scalability w.r.t. Source Document Size

We then performed a scalability test for our algorithms, to check how they perform on source documents of varying size. We have employed documents whose sizes ranged from $500KB$ to $50MB$, and have observed their performance, as shown in Figure 25 (a) for the PINT algorithm and in Figure 25 (b) for the PDDT algorithm (in both plots, the y axis is in logarithmic scale). The cost of updating the lattice (i.e., the auxiliary structures) is the most significant component of the view maintenance costs both for insertions and deletions, while the delta table computation and the time to get the update expression are comparably small. In Figure 25 (b), it can be noted that the time

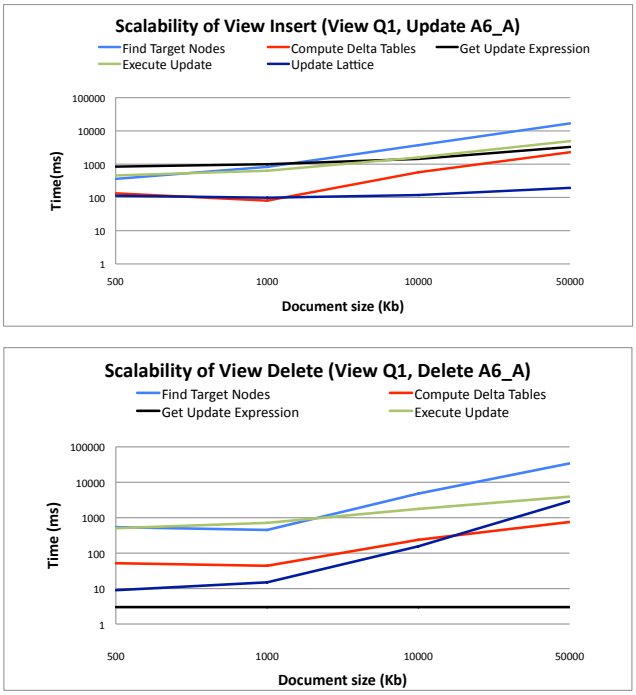


Fig. 25. Scalability for XMark view Q_1 and update $A6\_A$.

to *Get Update Expression* is negligible in the case of deletions due to the efficiency of the pruning methods. Moreover, the time to *Execute Update*, i.e., the time to compute the join expressions which determine which tuples should be added/removed to the view, respectively for the insertions/deletions, has a cost that gracefully grows with the size of the document, and has a similar trend of the time to *Find target nodes*.

In summary, this experiment shows that the cost for view maintenance is beneficial for all the document sizes up to $50MB$. Additional experiments shown below, comparing with full view recomputation will further confirm this claim.

6.5. Comparison with Full Recomputation

We have conducted an experiment to measure the gain that our incremental algorithms have over the baseline case, when the view is fully recomputed from the modified document. This experiment aimed to compare the time necessary to incrementally update a view, by exploiting snowcaps and pruning the term expression, with the time required to recalculate the view after a source update.

Figure 26 and Figure 27 report the results for each view-insert and view-delete pair. It can be seen that the full expression recomputation becomes prohibitive for many of the scenarios, while the incremental view maintenance achieves much lower times. The difference is even more remarkable for the deletion cases, as Figure 27 shows.

6.6. Comparison with Previous Algorithm

In order to show the benefit of bulk updates for incremental view maintenance, in our framework we have re-implemented IVMA, the view maintenance algorithm described in [Sawires et al. 2005]. Compared to the original version with a relational back-end, our own implementation of [Sawires et al. 2005] relies on a native XML store. This algorithm propagates XPath view updates which add or delete exactly one node at a time. For this experiment, we used insertions which add a fixed XML

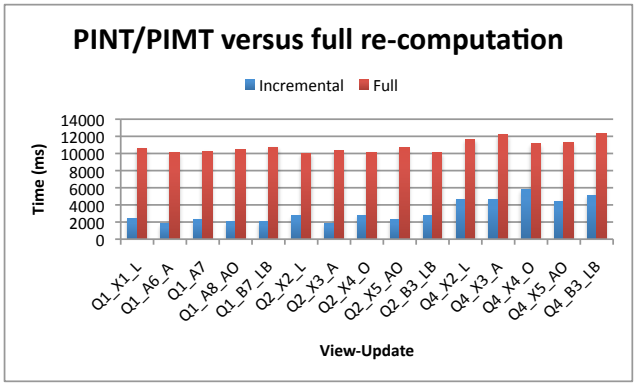


Fig. 26. Incremental maintenance vs. recomputation due to insertions, for the XMark views Q_1, Q_2 and Q_4.

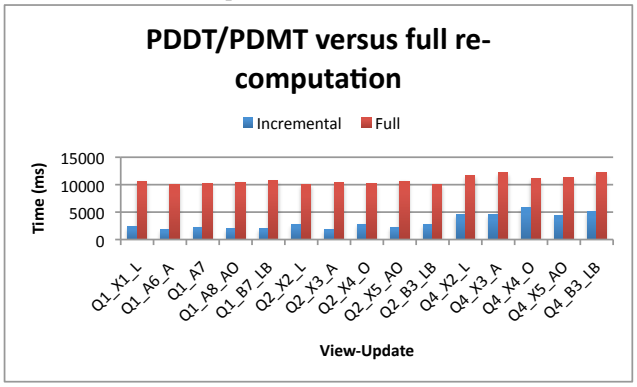


Fig. 27. Incremental maintenance vs. recomputation due to deletions, for the XMark views Q_1, Q_2 and Q_4.

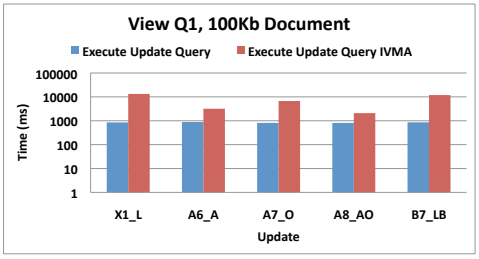


Fig. 28. PINT/PIMT algorithms compared with the IVMA algorithm [Sawires et al. 2005] for view Q_1.

tree, consisting of a root node with four children. Such an insertion is handled in one shot by our algorithm, and by five consecutive calls to IVMA [Sawires et al. 2005]. These experiments have been executed on a Linux 2.6.31.13-server-1mnb, with 2.33GHz Intel(R) Xeon(R) CPU 5140 and 4GB memory. Figure 28 (in logarithmic scale) shows that our approach outperforms IVMA by (at least) one order of magnitude for the view Q_1 and a source document of 100KB.

6.7. Impact of the Snowcaps Used During View Maintenance

In this section, we study the trade-offs between two simple alternatives of storing the lattice nodes. The first alternative, which we term *Snowcaps* relies on a set of snowcaps selected as follows. In a given lattice, we pick a small yet sufficient set of snowcaps required to maintain the view, more specifically, we pick one snowcap at each level. When several snowcaps exist at the same level (recall Figure 6 and 7), we simply pick the first. Obviously, *Snowcap* also maintains the lattice

leaves. In the second alternative, namely *Leaves*, we only store the lattice leaves and we compute the needed snowcaps on the fly. Obviously many other alternatives exist, and a cost-based choice would be needed to make the best choice, as we explained in Section 3.5. The choices we make here are for illustration.

We have measured two different times: *(R)* the time to evaluate the terms in the view expression by using the snowcaps or leaves and *(U)* the time to update the leaves and snowcaps after the update takes place.

In the first graphs (Figures 29 and 30), we focus on the total times to evaluate the terms and update the lattice for snowcap lattices and leaf lattices, for views Q4 and Q6 respectively. The performance is significantly better for the snowcaps lattice compared to the leaves lattice. The performance benefit for Q4 is less than that observed for Q6. This difference can be explained by the number of snowcaps and number of tuples contained in the view. The benefit decreases as the number of snowcaps and tuples increases, which explains the difference in performance for Q4 and Q6. Experiments run on the other queries of the benchmark (omitted for space reasons) confirmed this trend.

In the second graphs (Figure 31 and 32), we focus on *(R)* and *(U)* times and we also report their total. It can be noticed that the time to run plans and to update with *Snowcaps* is less than the corresponding time of the *Leaves* alternative. These results confirm the trend observed with the total time in the first graphs. The above experiments let us observe that for all the views investigated in our study, lattice snowcaps result in a better method for view materialization than lattice leaves and entire lattices.

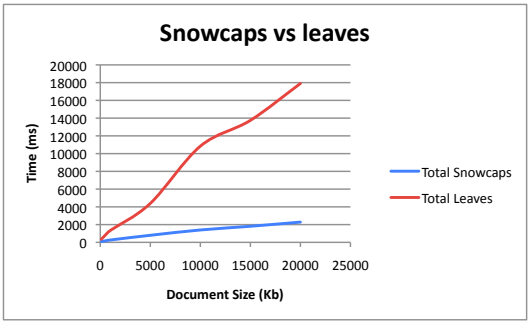


Fig. 29. Time to evaluate terms and update lattice for snowcaps vs leaves for the XMark view Q_4.

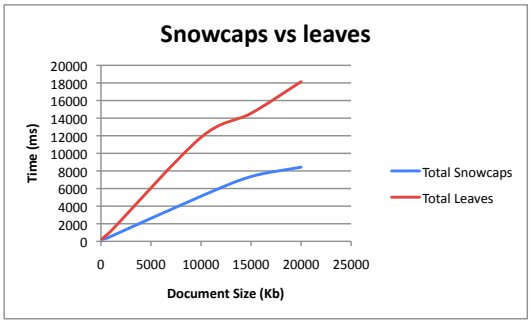


Fig. 30. Time to evaluate terms and update lattice for snowcaps vs leaves for the XMark view Q_6.

6.8. Optimisations

We ran experiments to study the benefits of the reduction rules presented in Section 5. As these rules are defined on atomic operations, we modified our system to operate in this manner. To test these rules we ran the update X1_L for XMark view Q1 simultaneously to a varying update that would have the same target nodes as a varying percentage of X1_L over a 100Kb database. These percentages were: 20%, 40%, 60%, 80%, and 100%. For example, to test rule O1 for the 20% test query X1_L (//site/people/person) was run alongside //site/people/person[profile]. This resulted in 30 target nodes with 5 duplicates. Therefore, by rule O1 these 5 duplicates were removed. O1 and O3 were both tested using deletions, whereas, I5 required insertions. The view and update were modified slightly for O3 in order to perform tests similar to the other two rules. We compared the performance of our view maintenance method with and without optimisation. The time for optimisation is included for the tests in which it is used. The results for O1, O3 and I5 can be seen in Figures 33, 34 and 35 respectively. We can see that the use of the optimisation rules improved the performance time, more so as the percentage increases.

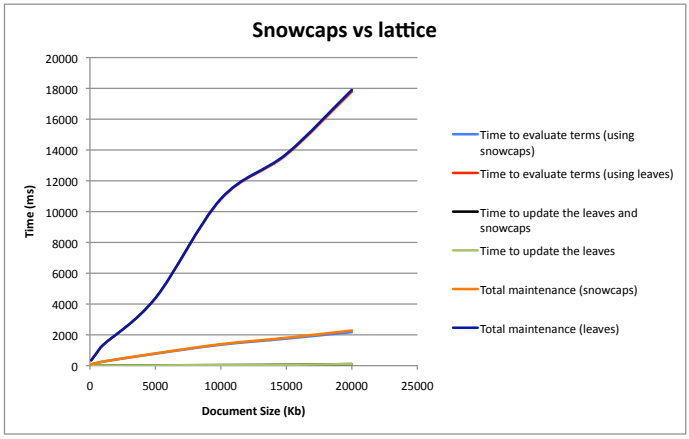


Fig. 31. Time to evaluate terms (R) and update snowcaps/leaves (U) and total maintenance time in the cases in which snowcaps, respectively, the lattice leaves are used for the XMark view Q_4.

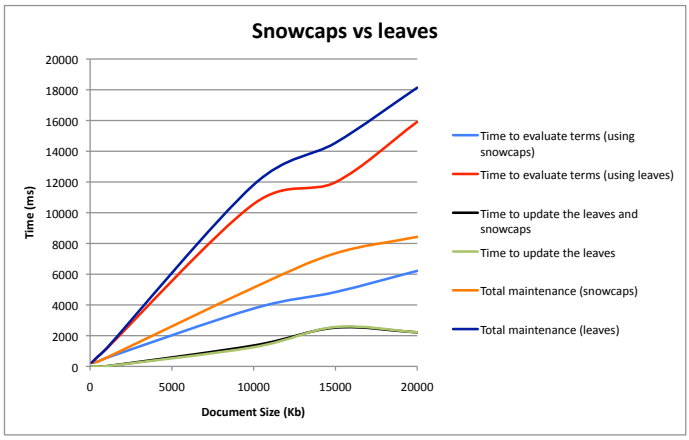


Fig. 32. Time to evaluate terms (R) and update snowcaps/leaves (U) and total maintenance time in the cases in which snowcaps, respectively, the lattice leaves are used for the XMark view Q_6.

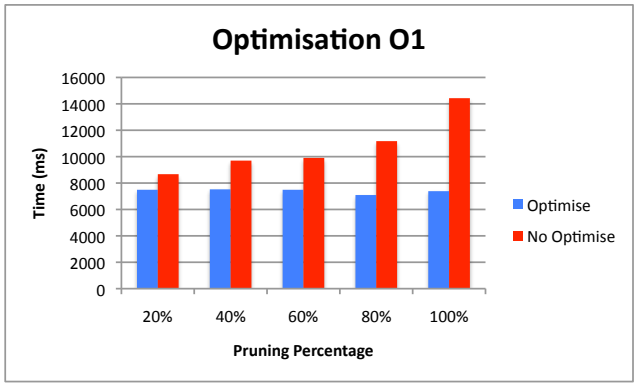


Fig. 33. Performance for reduction rule O1.

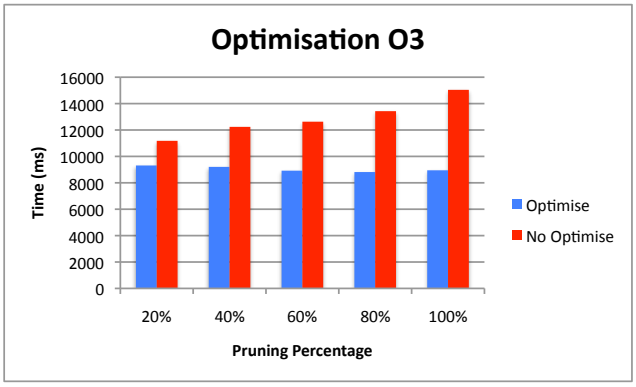


Fig. 34. Performance for reduction rule O3.

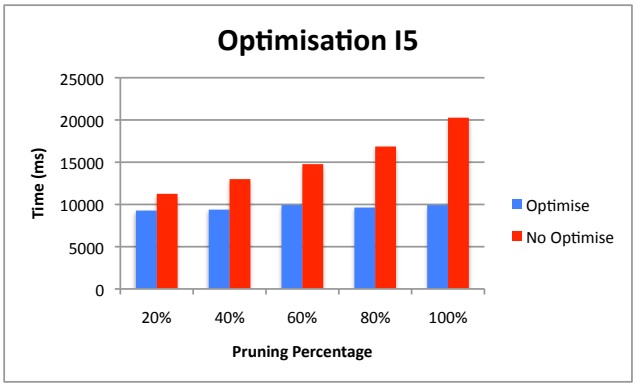


Fig. 35. Performance for reduction rule I5.

7. RELATED WORK

A large body of past research has been devoted to view updates in the context of relational databases [Bancilhon and Spyratos 1981; Bohannon et al. 2006; Gottlob et al. 1988; Gupta et al. 1993]. [Bancilhon and Spyratos 1981; Gottlob et al. 1988] focuses on *the view update problem*, i.e., on how to translate a view update into a database update, while avoiding the presence of inconsistencies and side effects on the view. Recently, [Bohannon et al. 2006] proposed *update policies*, expressed in a bidirectional language, to guarantee that the view update is well behaved and handles arbitrary changes to the view. Optimal incremental view maintenance algorithms for relational and deductive database systems were presented in [Gupta et al. 1993], where the notion of derivation count for each tuple in the view is introduced. The algorithms addressed consider both recursive and non-recursive views. In both cases, view definitions are used to generate a set of rules that compute the changes to the views using the base relations and the old views.

In the context of the XML data model, quite recently, [Björklund et al. 2009] studied the problem of incremental view maintenance in its Boolean version and with respect to the XPath language. Boolean incremental view maintenance checks that, after applying the update to the base data, the XPath expression representing the view is still satisfied. Similarly to our approach, they studied the above problem and derived its complexity *per update*, i.e., by considering one update at a time. The view language they consider is slightly more limited than ours (in particular, they do not support

multiple returning nodes), and their approach does not consider incorporating efficient XML query processing techniques in the view update process.

A practical fragment of XPath $\{//, /, *, []\}$ has been used in previous work on incremental view maintenance for XPath [Sawires et al. 2005; Sawires et al. 2006]. Interestingly, they also consider count() predicates, therefore view maintenance may be non-monotonic: adding some XML nodes may lead to removing data from the view, while removing XML nodes may add data to a view. In contrast, our conjunctive tree pattern dialect is monotonic. Compared to [Sawires et al. 2005], we focus on (i) tree patterns with multiple return nodes, which cannot be handled by the approach of [Sawires et al. 2005; Sawires et al. 2006] (based on the analysis of the XPath "view main path"). Views with multiple return nodes may lead to very efficient multiple-view rewritings [Manolescu et al. 2011]; and (ii) bulk updates, where several nodes can be added at the same time. As we have argued in the introduction, the XQuery Update language gives many opportunities for such updates. Moreover, our experiments demonstrated that our algorithms, leveraging state-of-the-art techniques in XML query evaluation, outperform repeated application of the node-based algorithm of [Sawires et al. 2005].

An extension of [Sawires et al. 2005] is [Sawires et al. 2006] which considers the case when the database and the view store are decoupled and the update has to be propagated using less information. The XPath dialect and node-at-a-time approach stay the same as in [Sawires et al. 2005].

To the best of our knowledge, the only works which study the incremental view maintenance problem for XQuery views are [Dimitrova et al. 2003; El-Sayed et al. 2006; Foster et al. 2008]. [Dimitrova et al. 2003; El-Sayed et al. 2006], focus on the maintenance of XQuery views over relational data. The algorithms of [Foster et al. 2008] translate updates through views expressed in the internal tree algebra of Galax. This approach is elegant due to its usage of an algebra. However, it requires that the view and the source document be kept in memory during the maintenance process. In contrast, our approach requires manipulating only tuples of IDs, that may be stored on disk (e.g. when storing snowcaps) and read as needed during view maintenance.

A close work in this area [Abiteboul et al. 2009] considers the maintenance of tree pattern views (with some non-monotonic extensions) over *active documents*, that is, XML documents including calls to Web services, which return streams of answers that are inserted in the documents. The solution consists of algorithms to be applied when each new answer is received and inserted in the document, it is a hybrid granularity between node-level (since an answer can contain several nodes) and statement level (since they do not use declarative update statements). Their solution relies on Datalog optimization techniques and on an XML database used as a black box, whereas we describe algorithms to be implemented inside the engine.

An important line of related works seek to identify when a view is unaffected by an update [Benedikt et al. 2005; Benedikt and Cheney 2010; Bidoit et al. 2010]. In contrast, we provide algorithms for propagating the insertions when the view is affected.

Finally, [Balmin et al. 2004a] also uses patterns for query processing based on materialized views. Our patterns have two significant differences from those in [Balmin et al. 2004a]. They consider node IDs are pointers into a store and therefore when using their views, there is always the possibility to access the data, whereas in our approach node IDs (and the views containing them) are standalone and we do not consider accessing the base data. They consider XPath views returning data from one node, whereas we consider richer tree patterns returning data from several nodes.

8. CONCLUSION AND FUTURE WORK

In this paper, we have devised algebraic incremental view maintenance algorithms, that work on a per-statement basis, as opposed to previous per-node approaches. By leveraging structural IDs and appropriate auxiliary data structures, our technique is efficient and scalable at the same time. With respect to our previous work [Anonymous], this paper has provided detailed propagation techniques for more XML update operations, such as the deletes, and studied their performance. We have also shown how our techniques can be combined efficiently with previous optimizations in order to speed up the application of sequences of XML updates, to views. updates propagated to

views. Future work is devoted to study a more powerful fragment of the view language and further optimization strategies for propagating update sequences to XQuery views.

A. TEST SET

We have used these updates to test the different kinds of XPath expressions that could be present in identifying the target node(s) for deletions or insertions. These updates were largely inspired by the XPathMark benchmark [Franceschet 2005] and the views (based on the XMark benchmark [Schmidt et al. 2002]) were created such that they would be affected by these updates. These updates are the expressions that can be represented within XML access modules [Arion et al. 2005] which are used to represent views and the location of target node(s). The following kinds of XPath expressions are supported:

— L: Linear path expression
— LB: Linear with Boolean filter
— A: AND predicate (pipeline the filters)
— O: OR predicate (union the paths)
— AO: AND + OR predicate (combination of the two former cases)

A.1. Linear Path Expression Updates

X1_L: insert_name
– For each person add a new name

```
let $c:= doc (“auction.xml”)
for $person in $c/site/people/person
insert ⟨name⟩ Martin
          ⟨name⟩ and ⟨/name⟩
          ⟨name⟩ some ⟨/name⟩
          ⟨name⟩ test ⟨/name⟩
          ⟨name⟩ nodes ⟨/name⟩
       ⟨/name⟩
```

B3 _L: insert_increase
– For each bidder add a new increase

```
let $c:= doc (“auction.xml”)
for $bidder in $c//open_auction/bidder
insert ⟨increase⟩ inserted 100.00
          ⟨increase⟩ and ⟨/increase⟩
          ⟨increase⟩ some ⟨/increase⟩
          ⟨increase⟩ test ⟨/increase⟩
          ⟨increase⟩ nodes ⟨/increase⟩
       ⟨/increase⟩
```

B3 _L: insert_increase
– For each open auction add a new increase for each bidder

```
let $c:= doc (“auction.xml”)
for $increase in
   $c/site/open_auctions/open_auction/bidder
insert ⟨increase⟩ inserted 300.00
          ⟨increase⟩ and ⟨/increase⟩
          ⟨increase⟩ some ⟨/increase⟩
          ⟨increase⟩ test ⟨/increase⟩
          ⟨increase⟩ nodes ⟨/increase⟩
       ⟨/increase⟩
```

E6 _L: insert_item
– For each item insert a new item inside it

```
let $c:= doc (“auction.xml”)
for $item in $c/site/regions/*/item
insert ⟨item⟩
          ⟨location⟩ Unknown⟨/location⟩
          ⟨quantity⟩ 1⟨/quantity⟩
          ⟨name⟩ E6_L Item ⟨/name⟩
          ⟨payment⟩
             Creditcard, Personal Check, Cash
           ⟨/payment⟩
       ⟨/item⟩
```

X17 _L: insert_item
– For each item insert a new item inside it

```
let $c:= doc (“auction.xml”)
for $item in $c/site/regions//item
insert ⟨item⟩
          ⟨location⟩ Unknown⟨/location⟩
          ⟨quantity⟩ 1⟨/quantity⟩
          ⟨name⟩ X17_L Item ⟨/name⟩
          ⟨payment⟩
             Creditcard, Personal Check, Cash
           ⟨/payment⟩
          ⟨description⟩
             Test description
           ⟨/description⟩
       ⟨/item⟩
```

B5 _L: insert_item
– For each item add a new name into its current name

```
let $c:= doc (“auction.xml”)
for $item in $c/site/regions/*/item/name
insert ⟨item⟩
          ⟨location⟩ Unknown⟨/location⟩
          ⟨quantity⟩ 1⟨/quantity⟩
          ⟨name⟩ B5_LB Item ⟨/name⟩
          ⟨payment⟩
             Creditcard, Personal Check, Cash
           ⟨/payment⟩
       ⟨/item⟩
```

A.2. Linear with Boolean Filter Updates

B7 _LB: insert_name – For each person with a profile with a salary attribute add a new name	**B3 _LB: insert_name** – For each open auction with a reserve add a new name
let $c:= doc (“auction.xml”) for $name in $c//person[profile/@income] insert ⟨name⟩ Jim ⟨name⟩ and ⟨/name⟩ ⟨name⟩ some ⟨/name⟩ ⟨name⟩ test ⟨/name⟩ ⟨name⟩ nodes ⟨/name⟩ ⟨/name⟩	let $c:= doc (“auction.xml”) for $increase in $c/site/open_auctions /open_auction[reserve]/bidder insert ⟨increase⟩ inserted 4.50 ⟨increase⟩ and ⟨/increase⟩ ⟨increase⟩ some ⟨/increase⟩ ⟨increase⟩ test ⟨/increase⟩ ⟨increase⟩ nodes ⟨/increase⟩ ⟨/increase⟩
B5 _LB: insert_item – For each item insert a new item inside it	
let $c:= doc (“auction.xml”) for $item in $c/site/regions/*/item[name] insert ⟨item⟩ ⟨location⟩ Unknown⟨/location⟩ ⟨quantity⟩ 1⟨/quantity⟩ ⟨name⟩ B5_LB Item ⟨/name⟩ ⟨payment⟩ Creditcard, Personal Check, Cash ⟨/payment⟩ ⟨/item⟩	

A.3. AND Predicate Updates

A6 _A: insert_name – For each person with a profile with a gender and a profile with an age add a new name	**X3 _A: insert_increase** – For each open auction with privacy and a bidder add an increase
let $c:= doc (“auction.xml”) for $name in $c/site/people /person[phone and homepage] insert ⟨name⟩ Mimma ⟨name⟩ and ⟨/name⟩ ⟨name⟩ some ⟨/name⟩ ⟨name⟩ test ⟨/name⟩ ⟨name⟩ nodes ⟨/name⟩ ⟨/name⟩	let $c:= doc (“auction.xml”) for $increase in $c/site/open_auctions /open_auction[privacy and bidder]/bidder insert ⟨increase⟩ inserted 150.00 ⟨increase⟩ and ⟨/increase⟩ ⟨increase⟩ some ⟨/increase⟩ ⟨increase⟩ test ⟨/increase⟩ ⟨increase⟩ nodes ⟨/increase⟩ ⟨/increase⟩

B1 _A: insert_item – For each item from North America or South America insert a new item inside it. let $c:= doc (“auction.xml”) for $item in $c/site /regions[namerica or samerica]//item insert ⟨item⟩ ⟨location⟩ Canada⟨/location⟩ ⟨quantity⟩ 1⟨/quantity⟩ ⟨name⟩ B1_A Item ⟨/name⟩ ⟨payment⟩ Creditcard, Personal Check, Cash ⟨/payment⟩ ⟨/item⟩	**E6 _A: insert_item** – For each item with a description and a name insert a new item inside it let $c:= doc (“auction.xml”) for $item in $c/site/regions/* /item[description][name] insert ⟨item⟩ ⟨location⟩ Unknown⟨/location⟩ ⟨quantity⟩ 1⟨/quantity⟩ ⟨name⟩ E6_L Item ⟨/name⟩ ⟨payment⟩ Creditcard, Personal Check, Cash ⟨/payment⟩ ⟨/item⟩
X20 _A: insert_item – For each item with a description and a name insert a new item inside it	
let $c:= doc (“auction.xml”) for $item in $c/site/regions// item[description][name] insert ⟨item⟩ ⟨location⟩ Unknown⟨/location⟩ ⟨quantity⟩ 1⟨/quantity⟩ ⟨name⟩ X20_A Item ⟨/name⟩ ⟨payment⟩ Creditcard, Personal Check, Cash ⟨/payment⟩ ⟨description⟩ Test description ⟨/description⟩ ⟨/item⟩	

A.4. OR Predicate Updates

A7 _O: insert_name – For each person with a phone or homepage add a new name	**X4 _O: insert_increase** – For each open auction with a bidder or privacy add a new increase
let $c:= doc (“auction.xml”) for $name in $c/site/people /person[phone or homepage] insert ⟨name⟩ Ioana ⟨name⟩ and ⟨/name⟩ ⟨name⟩ some ⟨/name⟩ ⟨name⟩ test ⟨/name⟩ ⟨name⟩ nodes ⟨/name⟩ ⟨/name⟩	let $c:= doc (“auction.xml”) for $increase in $c/site/open_auctions /open_auction[bidder or privacy]/bidder insert ⟨increase⟩ inserted 200.00 ⟨increase⟩ and ⟨/increase⟩ ⟨increase⟩ some ⟨/increase⟩ ⟨increase⟩ test ⟨/increase⟩ ⟨increase⟩ nodes ⟨/increase⟩ ⟨/increase⟩

X7 _O: insert_item – For each item with a description or a name insert a new item inside it	**B1 _O: insert_item** – For each item with a description or a name insert a new item inside it
let $c:= doc (“auction.xml”) for $item in $c/site/regions //item[description or name] insert ⟨item⟩ ⟨location⟩ Unknown⟨/location⟩ ⟨quantity⟩ 1⟨/quantity⟩ ⟨name⟩ X7_O Item ⟨/name⟩ ⟨payment⟩ Creditcard, Personal Check, Cash ⟨/payment⟩ ⟨/item⟩	let $c:= doc (“auction.xml”) for $item in $c/site /regions[namerica or samerica]/item insert ⟨item⟩ ⟨location⟩ Canada ⟨/location⟩ ⟨quantity⟩ 1⟨/quantity⟩ ⟨name⟩ B1_O Item ⟨/name⟩ ⟨payment⟩ Creditcard, Personal Check, Cash ⟨/payment⟩ ⟨description⟩ Test description ⟨/description⟩ ⟨/item⟩

A.5. AND + OR Predicate Updates

A8 _AO: insert_name – For each person with an address AND (phone OR homepage) AND (creditcard OR profile)	**X5 _AO: insert_increase** – For each open auction with a current AND (bidder OR reserve) add a new increase
let $c:= doc (“auction.xml”) for $name in $c/site/people /person[address and (phone or homepage) and (creditcard or profile)] insert ⟨name⟩ Angela ⟨name⟩ and ⟨/name⟩ ⟨name⟩ some ⟨/name⟩ ⟨name⟩ test ⟨/name⟩ ⟨name⟩ nodes ⟨/name⟩ ⟨/name⟩	let $c:= doc (“auction.xml”) for $increase in $c/site/open_auctions /open_auction[current and (bidder or reserve)]/bidder insert ⟨increase⟩ inserted 250.00 ⟨increase⟩ and ⟨/increase⟩ ⟨increase⟩ some ⟨/increase⟩ ⟨increase⟩ test ⟨/increase⟩ ⟨increase⟩ nodes ⟨/increase⟩ ⟨/increase⟩
X8 _AO: insert_item – For each item with a description AND (name OR mailbox) insert a new item inside it let $c:= doc (“auction.xml”) for $item in $c/site/regions //item[description and (name or mailbox)] insert ⟨item⟩ ⟨location⟩ New Zealand⟨/location⟩ ⟨quantity⟩ 1⟨/quantity⟩ ⟨name⟩ X8_O Item ⟨/name⟩ ⟨payment⟩ Creditcard, Personal Check, Cash ⟨/payment⟩ ⟨/item⟩	

A.6. Views

Q1	Q2
let $auction := doc("auction.xml") return for $b in $auction/site/people/person[@id] return $b/name/text()	let $auction := doc("auction.xml") return for $b in $auction/site/open_auctions/open_auction return $b/bidder/increase
Q3	**Q4**
let $auction := doc("auction.xml") return for $b in $auction/site/open_auctions/open_auction where $b/bidder/increase/text() = "4.50" return $b/bidder/increase/text()	let $auction := doc("auction.xml") return for $b in $auction/site/open_auctions/open_auction where $b/bidder/personref[@person = "person12"] return $b/bidder/increase/text()
Q6	**Q13**
let $auction := doc("auction.xml") return for $b in $auction/site/regions return $b//item	let $auction := doc("auction.xml") return for $i in $auction/site/regions/namerica/item return $i/name/text(), $i/description
Q17	
let $auction := doc("auction.xml") return for $b in $auction/site/people/person[homepage] return $b/name/text()	

REFERENCES

ABITEBOUL, S., BOURHIS, P., AND MARINOIU, B. 2009. Efficient maintenance techniques for views over active documents. In *EDBT*.

AL-KHALIFA, S., JAGADISH, H. V., PATEL, J. M., WU, Y., KOUDAS, N., AND SRIVASTAVA, D. 2002. Structural joins: A primitive for efficient XML query pattern matching. In *ICDE*.

AMER-YAHIA, S., CHO, S., LAKSHMANAN, L. V. S., AND SRIVASTAVA, D. 2002. Tree pattern query minimization. *VLDB J. 11,* 4.

Anonymous. Short (conference) version of this paper, details omitted due to double-blind reviewing.

ARION, A., BENZAKEN, V., AND MANOLESCU, I. 2005. XML access modules: Towards physical data independence in XML databases. In *XIME-P*.

ARION, A., BENZAKEN, V., MANOLESCU, I., PAPAKONSTANTINOU, Y., AND VIJAY, R. 2006. Algebra-based identification of tree patterns in XQuery. In *FQAS*.

BALMIN, A., OZCAN, F., BEYER, K., COCHRANE, R., AND PIRAHESH, H. 2004a. A framework for using materialized XPath views in XML query processing. In *VLDB*.

BALMIN, A., PAPAKONSTANTINOU, Y., AND VIANU, V. 2004b. Incremental validation of XML documents. *ACM Trans. Database Syst. 29,* 4.

BANCILHON, F. AND SPYRATOS, N. 1981. Update semantics of relational views. *ACM Trans. Database Syst. 6,* 4.

BENEDIKT, M., BONIFATI, A., FLESCA, S., AND VYAS, A. 2005. Verification of tree updates for optimization. In *CAV*.

BENEDIKT, M. AND CHENEY, J. 2009. Schema-based independence analysis for XML updates. In *VLDB*.

BENEDIKT, M. AND CHENEY, J. 2010. Destabilizers and independence of XML updates. In *VLDB*.

BIDOIT, N., COLAZZO, D., AND ULLIANA, F. 2010. Detecting XML query-update independence. In Bases de Données Avancées 2010 *(informal proceedings)*.

BJÖRKLUND, H., GELADE, W., MARQUARDT, M., AND MARTENS, W. 2009. Incremental XPath evaluation. In *ICDT*.

BOHANNON, A., PIERCE, B. C., AND VAUGHAN, J. A. 2006. Relational lenses: a language for updatable views. In *PODS*.

CAVALIERI, F., GUERRINI, G., AND MESITI, M. 2011. Dynamic reasoning on XML updates. In *Proceedings of the 14th International Conference on Extending Database Technology*. 165–176.

CHEN, Y., DAVIDSON, S. B., AND ZHENG, Y. 2006. An efficient XPath query processor for XML streams. In *ICDE*.

CHOI, B., CONG, G., FAN, W., AND VIGLAS, S. 2008. Updating Recursive XML Views of Relations. *J. Comput. Sci. Technol. 23,* 4.

DIMITROVA, K., EL-SAYED, M., AND RUNDENSTEINER, E. A. 2003. Order-sensitive view maintenance of materialized XQuery views. In *ER*.

EL-SAYED, M., RUNDENSTEINER, E. A., AND MANI, M. 2006. Incremental maintenance of materialized XQuery views. In *ICDE*.

FOSTER, J. N., KONURU, R., SIMEON, J., AND VILLARD, L. 2008. An algebraic approach to view maintenance for XQuery. In *PLAN-X workshop*.

FRANCESCHET, M. 2005. XPathMark: An XPath benchmark for the XMark generated data. In *XSym*.

GEORGIADIS, H., CHARALAMBIDES, M., AND VASSALOS, V. 2009. Cost based plan selection for XPath. In *SIGMOD*. 603–614.

GOTTLOB, G., PAOLINI, P., AND ZICARI, R. 1988. Properties and update semantics of consistent views. *ACM TODS 13,* 4.

GUPTA, A., MUMICK, I. S., AND SUBRAHMANIAN, V. S. 1993. Maintaining views incrementally. In *SIGMOD*.

MANDHANI, B. AND SUCIU, D. 2005. Query caching and view selection for XML databases. In *VLDB*.

MANOLESCU, I., KARANASOS, K., VASSALOS, V., AND ZOUPANOS, S. 2011. Efficient XQuery rewriting using multiple views. In *ICDE*.

MANOLESCU, I. AND ZOUPANOS, S. 2009. XML materialized views in P2P. DataX workshop.

ONIZUKA, M., CHAN, F. Y., MICHIGAMI, R., AND HONISHI, T. 2005. Incremental maintenance for materialized XPath/XSLT views. In *WWW*.

ONOSE, N., DEUTSCH, A., PAPAKONSTANTINOU, Y., AND CURTMOLA, E. 2006. Rewriting nested XML queries using nested views. In *SIGMOD*.

POLYZOTIS, N. AND GAROFALAKIS, M. 2006. XSKETCH synopses for XML data graphs. *ACM Trans. Database Syst. 31,* 3, 1014–1063.

SAWIRES, A., TATEMURA, J., PO, O., AGRAWAL, D., ABBADI, A. E., AND CANDAN, K. S. 2006. Maintaining XPath views in loosely coupled systems. In *VLDB*.

SAWIRES, A., TATEMURA, J., PO, O., AGRAWAL, D., AND CANDAN, K. S. 2005. Incremental maintenance of path-expression views. In *SIGMOD*.

SCHMIDT, A., WAAS, F., KERSTEN, M. L., CAREY, M. J., MANOLESCU, I., AND BUSSE, R. 2002. XMark: A benchmark for XML data management. In *VLDB*.

TANG, N., YU, J. X., ÖZSU, M. T., CHOI, B., AND WONG, K.-F. 2008. Multiple materialized view selection for XPath query rewriting. In *ICDE*.

WU, Y., PATEL, J. M., AND JAGADISH, H. V. 2003. Structural join order selection for XML query optimization. In *ICDE*.

XPath 2.0 1999. XML Path Language. http://www.w3.org/TR/xpath/.

XQuery 1.0 2009. The XML Query Language. http://www.w3.org/XML/Query.

XQuery Data Model 2010. XQuery 1.0 and XPath 2.0 Data Model (XDM) (Second Edition). http://www.w3.org/TR/xpath-datamodel/.

XQuery Update Facility 1.0 2009. The XQuery Update Facility 1.0. http://www.w3.org/TR/2009/CR-xquery-update-10-20090609/.

XU, L., LING, T. W., WU, H., AND BAO, Z. 2009. DDE: from Dewey to a fully dynamic XML labeling scheme. In *SIGMOD*.

XU, W. AND OZSOYOGLU, M. 2005. Rewriting XPath queries using materialized views. In *VLDB*.

YU, P. S., CHEN, M.-S., HEISS, H.-U., AND LEE, S. 1992. On workload characterization of relational database environments. *IEEE Trans. Software Eng. 18,* 4, 347–355.